HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Evaluierung und konzeptioneller Vergleich der Complex Event Processing Engine Siddhi anhand Esper

Saskia Stenzel

Bachelor-Arbeit im Studiengang „Angewandte Informatik“

20. Dezember 2018

Autor: Saskia Stenzel
Matrikelnummer: 1381685
saskia-stenzel@gmx.de

Erstprüfer: Prof. Dr. Ralf Bruns
Abteilung Informatik, Fakultät IV
Hochschule Hannover
ralf.bruns@hs-hannover.de

Zweitprüfer: Prof. Dr. Jürgen Dunkel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
juergen.dunkel@hs-hannover.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Bachelor-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 20. Dezember 2018 Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Auflistung

Abkürzungsverzeichnis

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CEP Complex Event Processing

CQL Continuous Query Language

CSV Comma-Separated Value

DW Data-Warehouse

DOM Document Object Model

EQL Esper Query Language

EDA Event-Driven Architecture

EPA Event Processing Agent

EPL Event Processing Language

EPN Event Processing Network

EQL Esper Query Language

ESB Enterprise Service Bus

FIFO First-In-First-Out

JSON JavaScript Object Notation

JVM Java Virtual Machine

MOM Message Oriented Middleware

MQTT Message Queuing Telemetry Transport

POJO Plain Old Java Object

SDK Software Development Kit

SQL Structured Query Language

WSO2 CEP WSO2 Complex Event Processor

XML Extensible Markup Language

1. Einleitung

1.1. Motivation

Das schnelle Verarbeiten großer Datenmengen ist mittlerweile ein wesentlicher Bestandteil in vielen Wirtschaftszweigen, wie zum Beispiel der Finanz- und der Logistikbranche, und somit auch ein wichtiger Erfolgsindikator. Dabei ist es wichtig, dass eingehende Datenströme aus einer Vielzahl von verschiedenen Quellen (z. B. Sensoren oder Geschäftsprozessen) nicht auf langer Zeit persistiert, sondern schnellstmöglich analysiert und auf diese entsprechend reagiert wird. Diese Anforderung wird mithilfe der Softwaretechnologie *Complex Event Processing* (CEP) umgesetzt. Im Vergleich zu zentralen Datenbanksystemen wie Data-Warehouse (DW), die strategische Entscheidungen aus historischen Daten ableiten, werden mithilfe von CEP die Informationen aus heterogenen Datenströmen nahezu in Echtzeit erkannt, analysiert und eine direkte operative Entscheidung abgeleitet.

In der heutigen Zeit führt die Verarbeitung von immensen Datenmengen (Big Data) oft schon zu einigen Problemen, da Systeme durch das langfristige Speichern an ihre Grenzen stoßen und unter Umständen bei der Verarbeitung eine lange Latenzzeit mit sich ziehen. Eine fortschrittliche Lösung dieses Problems bietet somit CEP und wird mittlerweile in vielen Industriebereichen für verschiedene Anwendungsfälle eingesetzt, z. B.

- **Finanzen** zur Handelsanalyse und Betrugserkennung
- **Fluggesellschaften** zur Betriebsüberwachung
- **Gesundheitswesen** zur Patientenüberwachung
- **Energie und Telekommunikation** zur Ausfallerkennung

Der Begiff CEP wurde erstmals von David C. Luckham[1] in seinem Buch *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems* (2002) erwähnt und hat sich anschließend als Grundbegriff der kontinuierlichen und zeitnahen Datenverarbeitung etabliert. Die eintreffenden Daten eines Datenstroms werden in CEP als ein *Ereignis* (event) bezeichnet. Ein Ereignis wird als Geschehen betrachtet, das eine Zustandsänderung des Systems repräsentiert. [Luc02]

[1] Professor für Elektrotechnik an der Stanford University und bekannt als Urheber des Complex Event Processings

Eines der Ziele von CEP ist es, aus einfachen Ereignissen aggregierte, d. h. komplexe Ereignisse einer höheren Abstraktionsebene zu erzeugen, indem Berechnungen oder auch Korrelationen mit anderen Ereignissen durchgeführt werden. Des Weiteren lassen sich Trends (z. B. Kaufverhalten von Kunden) oder auch Auffälligkeiten (z. B. Kreditkartenbetrug) durch Mustererkennungen in kontinuierlich eintreffenden Ereignissen aufspüren. CEP ist daher als eine bedeutende Lösung zur Bewältigung von stetig ansteigenden Datenmengen in der Industrie herangewachsen.

Der Gebrauch von CEP erfordert entsprechende Komponenten, die auf Ereignisse reagieren und diese behandeln. Als Kernkomponente werden in verteilten Systemen sogenannte *CEP Engines* eingesetzt, die Ereignismuster in den Datenströmen bzw. *Ereignisströmen* (event stream) erkennen. CEP Engines nutzen eine *Ereignisanfragesprache* (Event Processing Language (EPL)), sodass der Benutzer eine *Ereignisregel* definiert, die permanent Ereignisse nach der festgelegten Bedingung auswertet. Diese wird beispielsweise an externe Anwendungsprogramme weitergeleitet, um weitere Aktionen auszulösen. Im Laufe der letzten Jahre hat sich eine große Reihe an verfügbaren CEP Engines von unterschiedlichen großen Softwareherstellern wie Oracle, TIBCO, IBM oder SAP angesammelt, sodass die Entscheidung für eine passende CEP Engine für ein verteiltes System schwerfällt. In dieser Arbeit wird die CEP Engine namens *Siddhi* vorgestellt, die als leichtgewichtige und leistungsstarke Engine mit zahlreichen Erweiterungen zur Verarbeitung von Ereignissen veröffentlicht wurde. Aus diesen Gründen soll Siddhi auf potenzielle Fähigkeiten in der Arbeit untersucht werden.

1.2. Ziel der Arbeit

Das Ziel der Bachelorarbeit ist die Evaluierung der Open-Source CEP Engine Siddhi. Für die Evaluierung soll ein ausgearbeiteter Kriterienkatalog dienen. Die Kriterien wurden dafür aus den konzeptionellen und sprachlichen Grundeigenschaften einer CEP Engine, als auch aus den allgemeinen Qualitätsanforderungen an eine Software auserwählt.

Um Siddhi anhand der Kriterien sinnvoll zu bewerten, wurde die etablierte CEP Engine *Esper* als direkter Vergleichskandidat herangezogen. Da Esper eine der weit verbreitetsten und vielseitigsten Open-Source CEP Engines ist und in vielen Unternehmen genutzt wird, darunter sechs der größten Softwareunternehmen wie beispielsweise Oracle [Esp18a], wurde sie für den Vergleich als angemessen befunden.

Am Ende der Arbeit soll die Bewertung des Vergleichs zwischen Siddhi und Esper tabellarisch zusammengefasst werden und eine anschließende Beurteilung mithilfe des resultierenden Ergebnis erfolgen, wann die Verwendung der CEP Engine Siddhi für empfehlenswert erscheint.

1.3. Aufbau der Arbeit

Die Arbeit teilt sich in fünf folgende Kapitel auf: Kapitel 2 umfasst zum Verständnis der Arbeit alle Grundlagen von CEP und behandelt darin die wichtigsten Grundbegriffe, den Aufbau einer ereignisorientierten Architektur sowie einen kurzen Überblick des Vergleichskandidaten Esper. Kapitel 3 führt in die primäre CEP Engine Siddhi ein und beschreibt größtenteils die Architektur der Engine. Kapitel 4 stellt die Fallstudie für Vergleichszwecke vor und beschreibt die Implementation der Fallstudie in Siddhi und Esper. Kapitel 5 listet alle Evaluierungskriterien auf und vergleicht jeweils die Engines anhand der aufgestellten Kriterien. Anschließend werden die resultierenden Auswertungen und das Endergebnis mithilfe einer Tabelle zusammengefasst. Kapitel 6 fasst das komplette Ergebnis der Arbeit zusammen und endet mit einer Empfehlung anhand der durchgeführten Evaluierung für oder gegen Siddhi.

2. Grundlagen von „Complex Event Processing“

In diesem Kapitel sollen die Grundlagen von Complex Event Processing anhand von Begriffsdefinitionen und einem Beispielszenario näher erklärt werden. Infolgedessen wird mittels der ereignisorientierten Architektur das Konzept und die Schichten sowie Komponenten von CEP vorgestellt. Zusätzlich gibt es einen kurzen Überblick der Vergleichsengine Esper, die zur späteren Evaluierung von Siddhi dient.

2.1. Motivation

Complex Event Processing ist als ein eigenständiges Themengebiet in den letzten Jahren herangewachsen und hat sich als wichtiger Bestandteil der Datenverarbeitung etabliert. Die Softwaretechnologie kombiniert Daten aus mehreren Quellen, um Ereignisse oder Muster abzuleiten, die auf komplizierte Umstände schließen lassen. Aus kontinuierlichen Datenströmen treffen ständig veränderbare Daten wie Sensordaten (z. B. Temperaturwerte, Vitalwerte) oder Marktdaten (z. B. Aktienpreise, Rohstoffpreise) ein und lassen sich als Ereignisse interpretieren. Aus diesem Ereignisstrom werden in Ereignisregeln festgelegte *Ereignismuster* (event pattern) erkannt, d. h. zum gesuchten Muster werden die eintreffenden Ereignisdaten extrahiert, aggregiert und korreliert. Um die Verarbeitungslogik der Ereignisregeln zu spezifizieren, verwendet CEP eine deklarative Ereignisanfragesprache. Tritt das letzte Ereignis ein, welches zur vollständigen Mustererkennung fehlt, kann eine entsprechende Reaktion nahezu in Echtzeit ausgelöst werden. Mögliche Reaktionen wären das Auslösen von Warnungen (z. B. beim Ausfall einer Produktionsmaschine) oder eine weitere Verarbeitung der resultierenden Ereignisse. [Hed17]

2.2. Einleitendes Beispielszenario

Um die folgenden Grundbegriffe von CEP besser zu verdeutlichen, soll ein kurzes Beispielszenario[1] vorgestellt werden. Anhand dessen sollen die Formulierung der Ereignisregeln und Erkennung von Ereignismustern genauer erläutert werden.

[1] Das Beispielszenario wurde aus [Hed17] entnommen und entsprechend weiter angepasst.

Das Szenario stellt eine Temperaturüberwachung im Smart Home dar, welches unvorhergesehene Probleme wie z. B. eine defekte Heizungsanlage oder einen starken Temperaturabfall durch vergessenes Schließen eines Fensters ermitteln soll. Die CEP Engine muss in der Lage sein, anhand von eintreffenden Ereignissen aus Temperatursensoren und speziellen Fenstersensoren ein Muster zu erkennen, welches die eben benannten Probleme erkennt und dementsprechend eine Warnnachricht erzeugt.

2.3. Grundbegriffe von CEP

2.3.1. Ereignisse

Ereignisse können sehr vielseitig verstanden werden. Ein Geschehen, eine Handlung oder ein Prozess kann als ein mögliches Ereignis gelten. Im CEP Spektrum werden Ereignisse als eine Zustandsänderung angesehen oder als ein Objekt, das eine Aufzeichnung einer Aktivität des Systems darstellt. [Robnd, Hed17] So ist die Veränderung einer Raumtemperatur, bezogen auf das Beispielszenario, ein potenzielles Ereignis. Ereignisse müssen nicht zwangsweise autonom sein, sondern können aus mehreren Ereignissen aggregiert, korreliert oder mit weiterem Kontextwissen (z. B. aus Datenbanken) angereichert werden. [BD15] In diesem Zusammenhang werden Ereignisse komplexer und werden deshalb, wie in Abbildung 2.1 dargestellt, auf verschiedene *Abstraktionsebenen* eingestuft. So bilden sich Ereignisse auf dem mittleren Abstraktionsgrad, die sich durch einfache Ereignisse zusammenfassen oder einen fachlichen Kontext bilden. [BD10]

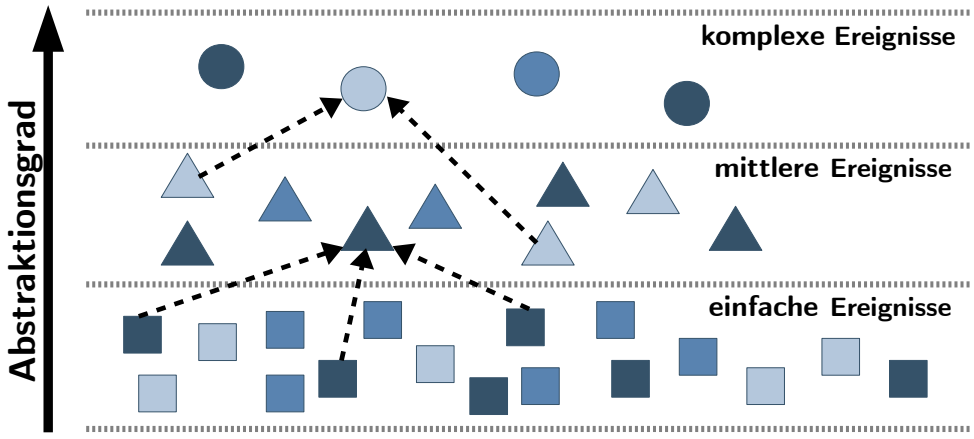


Abbildung 2.1.: Abstraktionsebenen von Ereignissen [vgl. BD10]

Im Vergleich zu objektorientierten Programmierparadigmen, benutzt CEP bei der Verarbeitung ebenfalls Klassen und Objekte von Ereignissen. [BD15] Diese sollen zunächst kurz erläutert werden:

- **Ereignistyp:** Ein *Ereignistyp* (event type) auch Ereignisklasse (event class) genannt, kennzeichnet die Ereignisse derselben Art mit ihren spezifischen Ereignisattributen (event properties). Relevante Ereignisattribute wären: eine eindeutige *Ereignisidentifikation* (event id), eine Auskunft über die Ereignisquelle und ggf. weitere charakteristische Attribute. Als wichtigster Kerninhalt der Ereignisse sind jedoch die Nutzdaten oder auch *Ereigniskontext* (event context) genannt. Sie beinhalten die geänderten Werte eines Zustands. [BD15]
- **Ereignisinstanz:** Eine *Ereignisinstanz* (event instance) oder auch Ereignisobjekt (event object) realisiert eine spezifische Ausprägung seines Ereignistyps. [BD15]

Zur Einfachheit wird in dieser Arbeit der Begriff Ereignis benutzt und nur an expliziten Stellen die beiden Begriffe Ereignistyp oder Ereignisinstanz verwendet.

Im Folgenden lassen sich die Ereignisse verschiedenartig klassifizieren.

Atomare Ereignisse

Bei einem *atomaren Ereignis* (auch primitives Ereignis genannt), handelt es sich um ein einfaches Ereignis, welches direkt aus einer *Ereignisquelle* (event source) stammt. Aus einer oder mehreren Ereignisquellen treffen die einfachen Ereignisse als ein Ereignisstrom ein, d. h. als eine kontinuierliche Sequenz aus jeweils unterschiedlichen Quellen. Diese rohen Ereignisse befinden sich dabei auf der niedrigsten Abstraktionsebene und besitzen im Einzelnen noch keine weitere Relevanz. Atomare Ereignisse treffen zu einem bestimmten Zeitpunkt ein und werden mit einem *Zeitstempel* (timestamp) versehen. [BD15]

Die Abbildung 2.2 zeigt ein beispielhaftes atomares Ereignis, welches von einem Temperatursensor mit einer Temperaturänderung von 23.4°C erzeugt wurde. Dieses erzeugte Ereignis ist eine spezifische Ausprägung seines Ereignistyps `Temperaturereignis` und wird deshalb als Ereignisinstanz bezeichnet. Es enthält dabei unterschiedliche Attribute mit unterschiedlichen Datentypen.

Ereignistyp:
`Temperaturereignis`
Ereignisidentifikation:
`283420`
Zeitstempel:
`2018-05-12 13:37:24`
Ereigniskontext:
`23.4`
Ereignisquelle:
`Temperatursensor_Wohnzimmer`
Ereignisname:
`Wohnzimmertemperatur`

Abbildung 2.2.: Ereignisinstanz einer Temperaturänderung [vgl. Hed17]

Komplexe Ereignisse

Komplexe Ereignisse formen sich aus mehreren atomaren Ereignissen, die in Beziehung stehen, sobald diese als Muster in einem Ereignisstrom erkannt werden. [BD10] Im Gegensatz zu atomaren Ereignissen bilden sie wie in Abbildung 2.1 zu sehen den höchsten Abstraktionsgrad und sind daher als fachliche Ereignisse (z. B. ein Geschäftsereignis „Ausverkauf eines Artikels“) einzustufen. [BD15] Komplexe Ereignisse beinhalten die Rückschlüsse und Erkenntnisse aus anderen Ereignissen und bestehen oftmals aus einer Menge von unterschiedlichen Ereignistypen. So würde sich beispielsweise eine Kombination aus mehreren Ereignissen eines Temperatursensors, die ein Muster bilden, wenn der Temperaturwert aller eintreffenden Temperaturereignissen kontinuierlich fällt. Die Situation würde auf eine defekte Heizungsanlage schließen und somit ein komplexes Ereignis abbilden. Dennoch kann es vorkommen, dass komplexe Ereignisse nur aus einer Menge von identischen Ereignismustern oder sogar auch nur einem einzigen atomaren Ereignis bestehen. [BD10, Hed17]

2.3.2. Ereignisregeln

Ereignisregeln sind der ausschlaggebende Kern von CEP und definieren die *Ereignisanfragen* (event query). Sie legen die Reaktion fest, die bei einer Mustererkennung im Ereignisstrom ausgelöst wird. Dieses Regeln bestehen aus einem *Bedingungsteil* und aus einem *Aktionsteil*. [BD15]

- **Bedingungsteil:** Der Bedingungsteil beschreibt das vorgegebene Ereignismuster, welches im Ereignisstrom erfasst werden soll. Sobald das Muster eingetroffen ist, wird der Aktionsteil der Regel angestoßen. [BD15]
- **Aktionsteil:** Der Aktionsteil gibt die Aktion vor, die bei einer erfüllten Bedingung ausgelöst wird. So könnte beispielsweise eine Nachricht verschickt oder ein weiterer Prozess in einer externen Anwendung angestoßen werden. Besonders wichtig ist die zusätzliche Generierung von neuen Ereignissen im Aktionsteil, die zur weiteren Verarbeitung genutzt werden. [BD15]

2.3.3. Ereignismuster und Kernoperatoren

Ein Ereignismuster steht für die Korrelationen zwischen Ereignissen in Ereignisströmen. Daher ist es wichtig, dass in Ereignisregeln temporale sowie kausale Beziehungen realisiert werden, um ein Ereignismuster zu definieren. [BD15]
Für solche Ereignismuster werden folgende Kernoperatoren in Ereignisregeln verwendet:

- **Konjunktionsoperator:** Eine *Konjunktion* A $\wedge$ B (auch A *AND* B) beschreibt, dass jeweils ein Ereignis vom Typ A und B eintreten muss. Die Reihenfolge, in der die Ereignisse eintreffen, ist dabei nicht entscheidend. [Hed17]

- **Disjunktionsoperator:** Eine *Disjunktion* A ∨ B (auch A *OR* B) beschreibt, dass mindestens ein Ereignis vom Typ A oder B eintreten muss. [Hed17]
- **Negationsoperator:** Eine *Negation* ¬A (auch *NOT* A) beschreibt, dass ein Ereignis vom Typ A nicht eintreffen darf. Hierbei unterscheidet man zwischen einer sequenzbasierten und einer fensterbasierten Negation. So dürfte beispielsweise bei der sequenzbasierten Negation kein Ereignistyp A zwischen der Sequenz zweier Ereignisse anderen Typs eintreffen. Bei der fensterbasierten hingegen dürfte kein Ereignistyp A in dem festgelegten Fensterintervall eintreffen. Mögliche Auswertungsfenster werden in Abschnitt 2.3.4 näher besprochen. [Hed17]
- **Sequenzoperator:** Eine *Sequenz* beschreibt die Abfolge in der die Ereignisse eintreffen müssen, d. h. die Ereignisse stehen zusammen in einer temporalen Beziehung. Die Sequenz A → B würde bedeuten, dass nach dem Ereignistyp A ein Ereignis von Typ B folgen muss. Innerhalb dieser Sequenz können durchaus andere Ereignistypen eintreffen, wenn dies durch ein Wiederholungsoperator definiert wird. [BD15]
- **Wiederholungsoperator (Kleene-(Plus/Stern)-Operator):** Eine *Wiederholung* beschreibt, dass ein Ereignistyp beliebig wiederholt werden kann. Mithilfe des Kleene-Plus-Operator (Kleene+) würde folgender Ausdruck A → B+ → C zwischen Ereignistyp A und C mindestens ein Ereignis vom Typ B erwarten. Die Wiederholungen können dementsprechend auch in einem festgelegten Fensterintervall definiert werden. [Hed17] Als weiterer Kleene-Operator gibt es den Kleene-Stern-Operator (Kleene*). Im Gegensatz zum Kleene-Plus-Operator würde der Ausdruck mit dem Kleene-Stern-Operator A → B* → C kein oder mehrere Ereignisse vom Typ B zwischen Ereignistyp A und C erwarten.
- **Aggregationsoperatoren (-funktionen):** Eine *Aggregation* beschreibt eine Zusammenfassung oder Anhäufung von Ereignissen anhand von gemeinsamen Eigenschaften z. B. Ereignisquelle oder Temperaturwert. In Abschnitt 2.3.1 wurden die möglichen Attribute eines Ereignisses erwähnt, mittels Aggregationsfunktionen können die Ereigniswerte dieser Attribute verarbeitet werden. Besonders zu beachten ist, dass diese vom gleichen Attributtyp (z. B. String, Integer, Double etc) sein müssen. [BD15]
  - **`sum()`:** Summiert die Werte des untersuchten Attributtyps aller Ereignisse auf. [Hed17]
  - **`avg()`:** Berechnet den Durchschnittswert des untersuchten Attributtyps aller Ereignisse. [Hed17]
  - **`min()/max()`:** Liefert den Minimal- bzw. Maximalwert des untersuchten Attributtyps aller Ereignisse zurück. [Hed17]

  Viele CEP-Systeme unterstützen weitere Aggregationsfunktionen oder auch die Möglichkeit, selbst Aggregationsfunktionen zu definieren. Eine Aggregation wird oftmals auch mit Vergleichsoperatoren (z. B. `==`, `!=`, `<`, `>`, `<=`, `>=`) für numerische

Attribute oder mit den bereits oben erwähnten logischen Operatoren (z.B. `AND`, `NOT`, `OR`) für boolesche Attribute dargestellt. [BD15, Hed17]

Eine Verschachtlung oder Mischform aus den oben genannten Kernoperatoren sind bei der Modellierung von Ereignismustern möglich. [Hed17]

Folgendes Ereignismuster des Temperaturüberwachungsszenario repräsentiert eine Zusammensetzung aus den eben vorgestellten Kernoperatoren:

```
WindowEvent(F=="zu") → not WindowEvent(F=="auf") → TempEvent(Temp < 10)
```

Das Ereignismuster enthält zwei Ereignistypen, einmal das `WindowEvent` für alle eintreffenden Fensterereignisse eines Fenstersensors und das `TempEvent` (abgek. TemperatureEvent) für alle eintreffenden Temperaturereignisse des Temperatursensors. Das Attribut `F` des `WindowEvents` steht für den Zustand des Fensters (auf oder zu) und das Attribut `Temp` des `TempEvents` für den gemessenen Temperaturwert (in °C). Die Pfeile zwischen den Teilausdrücken werden hier als eine zeitliche Sequenz von Ereignissen interpretiert. Der erste Teilausdruck verlangt das Eintreffen von `WindowEvents`, die ein geschlossenes Fenster repräsentieren. Der dritte Teilausdruck erwartet `TempEvents`, die einen Temperaturwert kleiner als 10°C aufweisen. Der zweite Teilausdruck hingegen verdeutlicht, dass zwischen diesen Ereignissen kein `WindowEvent` mit einem offenen Fenster auftreten darf. Eine gültige Mustererkennung mit der Ereignisabfolge wäre z. B. [`WE(F="zu"); TE(Temp=23); TE(Temp=16); WE(F="zu"); TE(Temp=12); TE(Temp=9); TE(Temp=7)`] mit `WE` für `WindowEvent` und `TE` für `TempEvent`. Wird dieses Muster erkannt, so lässt sich daraus schließen, dass die Heizungsanlage defekt ist und eine sofortige Reaktion (z. B. Warnnachricht an den Hausbesitzer) kann ausgelöst werden. [Hed17]

2.3.4. Windows

In CEP gibt es die Möglichkeit, mit Windows (z. Dt. Fenster) nur einen bestimmten Anteil einer Ereignismenge zu betrachten. Diese Windows werden ebenfalls in den Ereignisregeln definiert. Im Folgenden werden zwei typische Varianten von Windows vorgestellt.

- Die **Sliding Windows** sind die meist genutzte Variante zur Auswertung von Ereignissen. Hierbei kommen die sogenannten *Zeitfenster* (time window) oder *Längenfenster* (length window) zum Einsatz. Bei einem Zeitfenster wird ein definiertes Zeitintervall genutzt, wohingegen beim Längenfenster nur eine maximale Anzahl an Ereignissen aufgenommen wird. Jedes erzeugte Window wird als eine *Fensterinstanz* (window instance) verstanden. Mithilfe eines festgelegten Verschiebefaktors können die Fensterinstanzen versetzt werden. Dieser Verschiebefaktor und die

Größe (Zeit/Länge) einer Fensterinstanz bestimmen, wann immer ein Ereignis in das Window aufgenommen oder entfernt wird. Die Fensterinstanzen können dabei in verschiedenen Beziehungen zueinander stehen und besitzen spezifische Namen. [Hed17]

  - **Rolling Window:** Überlappen sich Fensterinstanzen, ist der Verschiebefaktor kleiner als die Fenstergröße. Diese Windows werden auch als sogenannte *rolling windows* bezeichnet. [Hed17]
  - **Tumbling Window:** Folgen Fensterinstanzen aufeinander oder zwischen ihnen besteht ein gewisser Abstand, dann ist der Verschiebefaktor größer oder gleich der Fenstergröße. Diese Windows werden auch *tumbling windows* genannt. [Hed17]

- Die **Landmark Windows** verwenden keinen Verschiebefaktor, sondern beinhalten ein festen Endzeitpunkt, an dem das Window geschlossen wird. So würden beispielsweise alle Ereignisse bis zum Zeitpunkt „jeden Montag 6 Uhr“ gesammelt werden und das Window würde wiederholt eine neue Fensterinstanz starten. [Hed17]

Im Folgenden sollen beispielhafte Abläufe für Sliding Windows erklärt werden und den Unterschied zwischen Tumbling Windows und Rolling Windows verdeutlichen. In Abschnitt 2.3.1 wurde erklärt wie Ereignisinstanzen eines Ereignistyps aufgebaut sind. Für das Beispiel der Sliding Windows sollen nun die Ereignisinstanzen eines Temperatursensors genutzt werden, die sich als geordnete Abfolge $\texttt{te}_1\texttt{; te}_2\texttt{; te}_3\texttt{; ...}$ darstellen lässt, d. h. $\texttt{te}_1$ ist die erste eintreffende Instanz vom Ereignistyp Temperaturänderung.

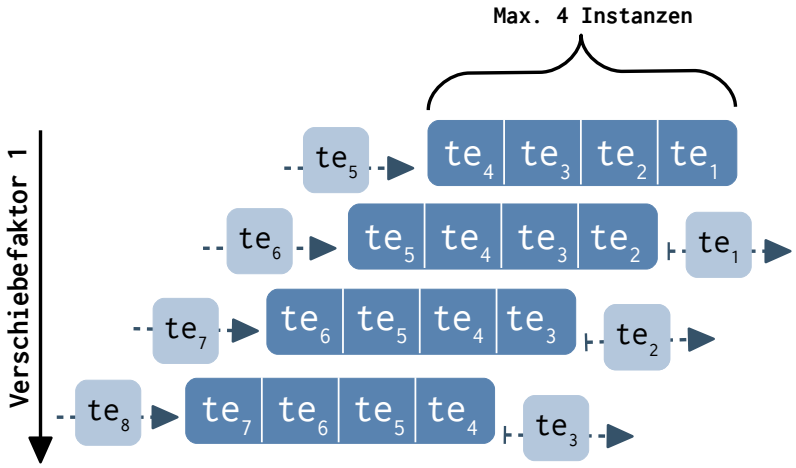


Abbildung 2.3.: Vier Fensterinstanzen eines Rolling Window mit dem Verschiebefaktor eins [vgl. Hed17]

Die Abbildung 2.3 zeigt ein Rolling Window, welches ein Längenfenster von vier Ereignisinstanzen und einem Verschiebefaktor von eins darstellt. Sobald eine neue Ereignis-

instanz eintrifft, wird die älteste Instanz nach dem FIFO-Prinzip (First-In-First-Out) aus dem Längenfenster ausgeschlossen und nicht mehr weiter betrachtet.

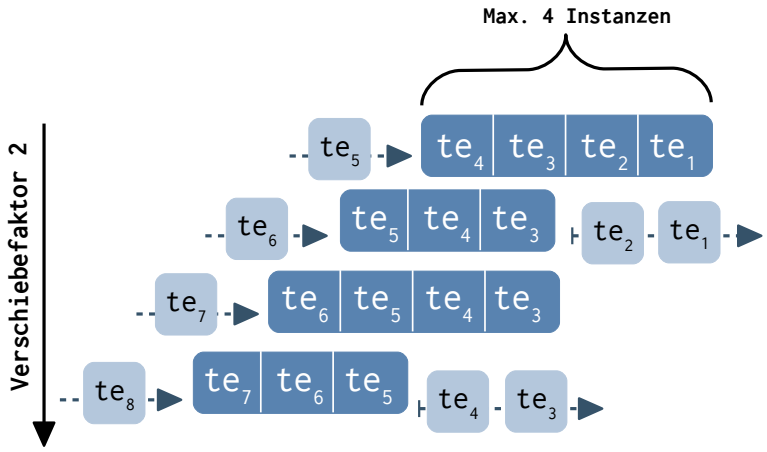


Abbildung 2.4.: Vier Fensterinstanzen eines Rolling Window mit dem Verschiebefaktor zwei [vgl. Hed17]

In Abbildung 2.4 hingegen werden mit einem Verschiebefaktor von zwei, die letzten beiden Ereignisinstanzen entfernt, sobald das Längenfenster die maximale Anzahl an Instanzen erreicht hat. Anschließend wird das Längenfenster wieder mit dem nächsten zwei eintreffenden Ereignisinstanzen aufgefüllt und die letzten beiden Instanzen werden wieder entfernt. Anstatt ein Längenfenster zu nutzen, lässt sich auch beispielsweise ein Zeitfenster mit der Dauer von 20 Minuten und einem Verschiebefaktor von fünf Minuten wählen. Dadurch werden jede fünf Minuten alle Ereignisse gelöscht, die älter als 15 Minuten sind.

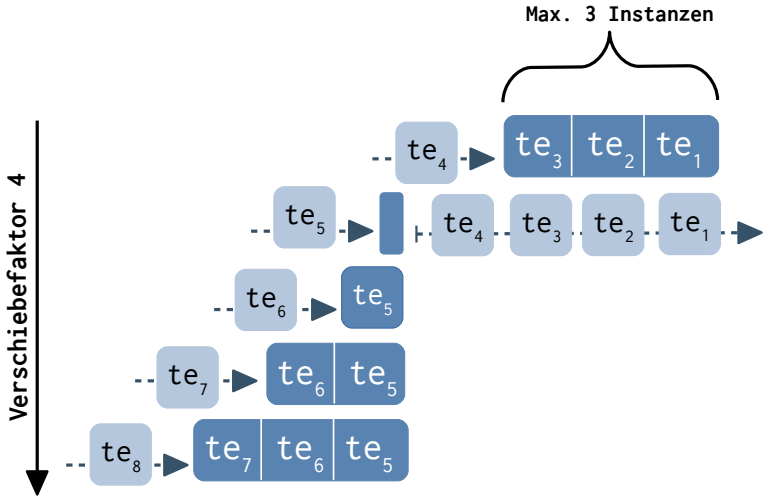


Abbildung 2.5.: Vier Fensterinstanzen eines Tumbling Window mit dem Verschiebefaktor vier [vgl. Hed17]

Wie bereits erwähnt, folgen die Fensterinstanzen bei einem Tumbling Window aufeinander oder es besteht ein gewisser Abstand zwischen den Fensterinstanzen. In Abbildung 2.5 wird durch den Verschiebefaktor von vier zwischen den Fensterinstanzen jeweils ein Abstand von einer Instanz gelassen. Sobald der Puffer von drei Instanzen voll ist, wird dieser komplett geleert und eine neu eintreffende Instanz wird ignoriert. Anschließend werden alle danach folgenden Instanzen wieder in das Längenfenster aufgenommen.

Sollen alle ankommenden Ereignisse berücksichtigt werden, ist es sinnvoll eine große Fensterlänge bzw. Fensterdauer mit einem möglichst kleinen Verschiebefaktor zu wählen. Dadurch werden einige Ereignisse zwar doppelt ausgewertet, aber grenzkritische Ereignisse können so nicht unbeabsichtigt ausgelassen werden. [Hed17]

2.3.5. Ereignisanfragesprache

Zur Formulierung von Ereignisregeln wird eine deklarative Ereignisanfragesprache verwendet. Anders als bei Datenbanken, die Anfragen gegen eine endliche Datenmenge durchführen, müssen Ereignisanfragen in der Lage, sein einen kontinuierlichen Datenstrom auszuwerten. Aus diesem Grund müssen Ereignisanfragesprachen einige Anforderungen bewältigen wie:

- **Datenextraktion:** Notwendige Daten müssen aus eintreffenden Ereignissen herausgezogen oder mit anderem Kontextwissen aus z. B. Datenbanken angereichert werden. Des Weiteren muss die Möglichkeit bestehen, die Datenstruktur, in denen die Ereignisse eintreffen oder ausgeliefert werden, zu transformieren. [EBle]
- **Komposition:** Einzelne Ereignisse nach datenbezogenen Zusammenhängen verbinden, sodass komplexere Ereignisse entstehen.
- **Akkumulation:** Ereignisanfragen mit Negationen oder Aggregationen auf eine begrenzte Ereignismenge mithilfe von Windows anwenden. [EBle]
- **Temporale Beziehungen:** Eine Ereignisanfrage sollte zeitliche Beziehungen von Ereignissen auswerten können, z. B. eine Sequenz: nach Ereignis A muss ein Ereignis vom Typ B eintreffen. [EBle]

In CEP haben sich drei Sprachstile zur Modellierung von Ereignisanfragesprachen angesammelt, die im Einzelnen erklärt werden.

- **Kompositionsoperatoren:** Die Kompositionsoperator-basierte Sprache stellt eine *Markup Language* (Auszeichnungssprache) für CEP dar. Mithilfe von Konjunktions-, Disjunktions-, Negations- oder Sequenzoperatoren (in Abschnitt 2.3.3 vorgestellt) werden die Ereignisanfragen realisiert. So würden beispielsweise bei einer Ereignisregel mit einem Konjunktionsoperator nach einem Muster suchen, dass eine bestimmten Kombination von Ereignissen repräsentiert, und ein neues Ereignis daraus generieren. Dennoch hat die Sprache ihre Schwächen, so kann beispielsweise

eine Anfrage mit schlechter Klammerung einer Sequenz unterschiedlich interpretiert werden und eine Aggregation sowie Komposition von Ereignisdaten lassen sich nur schwierig umsetzen. [EBle]

- **Datenstrom-Anfragesprachen:** Einer der weit verbreitetsten Datenstrom-Anfragesprachen ist die *Continuous Query Language* (CQL), welche auf der Datenbanksprache Structured Query Language (SQL) basiert. Anders als bei SQL-Anfragen auf Datenbanken, werden die CQL-Anfragen auf einen kontinuierlichen Ereignisstrom angewendet. Die Ereignisanfragen werden ebenfalls wie bei der Kompositionsoperator-basierte Sprache mithilfe von Operatoren (Konjunktion, Disjunktion, Negation und Sequenz, siehe Abschnitt 2.3.3) umgesetzt. Zusätzlich können in den Anfragen auch Aggregationsfunktionen und Auswahlstrategien (z. B. Windows, siehe Abschnitt 2.3.4) genutzt werden. Die CQL-basierte Datenstrom-Anfragesprache ist in CEP die meist genutzte und effizienteste Ereignisanfragesprache. [EBle]
- **Produktionsregeln:** Die Produktionsregeln werden im Wesentlichen mit einer Wirtsprogrammiersprache wie Java umgesetzt. Im Gegensatz zu den anderen beiden Sprachstilen sind die Regeln zustandsorientiert, d. h. Aktionen werden ausgeführt sobald eine bestimmte Menge an Ereigniszuständen eingetroffen ist. Diese Ereigniszustände, auch Fakten genannt, legen somit die Beziehungen zwischen Ereignisse fest. So muss für jedes Ereignis ein entsprechendes Faktum generiert werden. Die Fakten werden anschließend als Objekte im Speicher, namens *Working Memory* abgelegt. Jedoch bringen Produktionsregeln zusätzlichen Aufwand mit sich. Es muss sich um das Entfernen der Fakten aus dem Working Memory selbst gekümmert werden. [EBle]

2.4. Ereignisgesteuerte Architektur

Eine ereignisgesteuerte Architektur (event-driven architecture (EDA)) beschreibt einen Softwarearchitekturstil und realisiert ein Systems, welches Interaktionen durch Ereignisse durchführt, d.h. die Kommunikation der internen Schichten und Komponenten erfolgt allein durch Ereignisaustausch. So werden Prozessabläufe anhand von eintreffenden Ereignissen aus internen oder externen Quellen ausgelöst. CEP ist die Verarbeitungsfunktion innerhalb einer EDA und ist für die Analyse und Verarbeitung der Ereignisse zwischen den Endpunkten (Ereignisquelle und Ereignissenke) einer EDA zuständig. Die Grundpipeline eines ereignisgesteuerten Systems lässt sich dabei folgendermaßen definieren: 1. Erkennen → 2. Verarbeiten → 3. Reagieren. Durch die Verwendung von einer ereignisgesteuerten Architektur werden erstrebenswerte Architekturprinzipien wie Flexibilität und Effizienz erfüllt. [BD10] Wodurch diese Architekturprinzipien umgesetzt werden, wird in Abschnitt 2.4.2 erklärt.

Im Folgenden sollen auf die Schichten und Komponenten der EDA eingegangen werden.

2.4.1. Schichten einer EDA

In einer EDA durchlaufen Ereignisse in der Regel drei Schichten, durch welche sie erzeugt, verarbeitet und behandelt werden. Als Nächstes sollen die drei Schichten, die in Abbildung 2.6 dargestellt sind, vorgestellt werden.

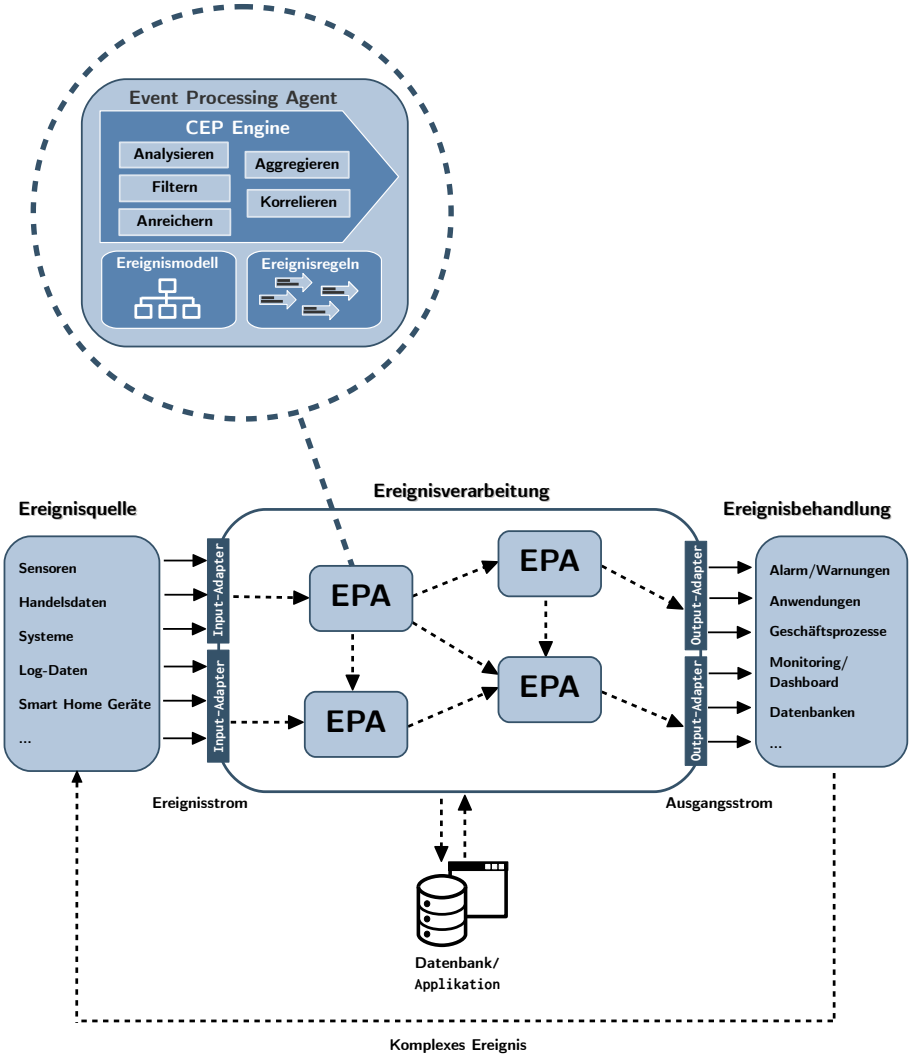


Abbildung 2.6.: Schichten und Komponenten einer ereignisgesteuerten Architektur [vgl. BD10]

Ereignisquelle

Die Aufgabe einer Ereignisquelle (z. B. Sensoren, Web Services, Smart Home Geräte, etc.) ist das Erzeugen von Ereignisinstanzen aus erfassten Nutzdaten. Diese werden über einen Input Adapter an die nächste Ereignisschicht geschickt und von der CEP Engine verarbeitet. [BD10]

Ereignisverarbeitung

In dieser Schicht findet die Verarbeitung der Ereignisse statt. Durch kontinuierlich eingehende Ereignisströme aus unterschiedlichen Ereignisquellen werden die einfachen Ereignisse von CEP Engines analysiert und zu komplexeren Ereignissen abgeleitet. Die konsumierten Ereignisse müssen oftmals in ein anderes Format umgewandelt werden, z. B. von JSON in XML. Außerdem kann es vorkommen, dass fehlerhafte Daten in den Ereignissen entstehen, sodass diese vorerst korrigiert werden müssen. In der Ereignisverarbeitung findet auch die Anreicherung mithilfe von Kontextwissen aus einer Datenbank statt, wenn die vorhandenen Nutzdaten der Ereignisse nicht ausreichen. [BD10]

Ereignisbehandlung

Treffen Ereignisse in die letzte Schicht ein, so lassen sich diese auf unterschiedliche Weise behandeln. Eine Ereignisbehandlung könnte demnach eine auszulösende Reaktion sein, z. B. die aktuelle Anzeige des Aktienkurs über ein Dashboard oder das Versenden einer Warnmeldung an mobile Endgeräte in kritischen Notfällen. Aber auch das bloße Speichern oder Anzeigen von Ereignissen wäre denkbar. Darüber hinaus kann die Ereignisbehandlungsschicht selbst als eine Ereignisquelle dienen und weitere komplexe Ereignisse erzeugen. [BD10]

Damit die Ereignisschichten untereinander kommunizieren können, werden diese über die notwendigen *Mediatoren* verbunden. Der Mediator allein verfügt über das Wissen, ob die Komponenten einer Ereignisschicht verfügbar sind oder überhaupt existieren. Zur Umsetzung eines Mediators wird eine Message Oriented Middleware (MOM) genutzt, welche als Kommunikationsmodell das Publish&Subscribe-Prinzips verwendet. Der Publisher (Sender) ist in diesem Fall die Ereignisquelle und veröffentlicht Ereignisse an mehrere CEP Engines, welche die Subscriber (Abonnenten) darstellen. Damit die Subscriber die Ereignisse empfangen, muss der Publisher ein *Topic* (Thema) im MOM generieren, sodass Subscriber bei Interesse das Topic abonnieren können. Durch den Einsatz dieser Kommunikationsmodells lassen sich, wie vorhin erwähnt, erstrebenswerte Architekturprinzipien erzielen. Durch die asynchrone Kommunikation kann das System trotz Ausfall einer Komponente die Verarbeitung weiter fortsetzen, ohne das Gesamtsystem zu beeinträchtigen. Außerdem verfügt das System über eine hohe Skalierbarkeit, da Komponenten untereinander lose gekoppelt sind. [BD10]

2.4.2. Komponenten der EDA

Event Processing Agent

Das Zentrum einer EDA bildet die CEP Komponente, auch *Event Processing Agent* (EPA) genannt. Die Hauptaufgabe der EPAs ist, die eintreffenden Ereignisse aufzunehmen, diese zu analysieren und zu verarbeiten. Nach der Verarbeitung erzeugt ein EPA als Ausgabe ein oder mehrere neue Ereignisse. Ein EPA beinhaltet üblicherweise diese drei Grundelemente: das Ereignismodell, die Ereignisregeln und die CEP Engine. Das Ereignismodell enthält dabei das komplette Wissen der eintreffenden Ereignisse wie Ereignistypen mit deren Attributen, Eigenschaften sowie Beziehungen und Abhängigkeiten zwischen den Ereignistypen. Erst durch die Spezifizierung von Ereignismodellen mit den entsprechenden Ereignisformaten können Ereignisse angemessen verarbeitet werden. [BD10] In Kapitel 4 wird auf solch ein Ereignismodell der Fallstudie eingegangen.

CEP Engine

Die Complex Event Processing Engine sitzt als wichtigster Kern im EPA. Ihre Aufgabe in einer EDA ist es, die eintreffenden Ereignisse mit definierten Mustern einer Regelmenge abzugleichen und anschließend bei einer Mustererkennung die festgelegte Reaktion in der Ereignisregel auszulösen. In diesem Zusammenhang kann es passieren, dass die CEP Engine auf vergangene Ereignisse zurückgreifen muss. Diese Ereignisdaten werden daher im Arbeitsspeicher bereitgehalten. Um nicht die komplette Menge an Ereignisse zu speichern, wird diese anhand eines Windows beschränkt, welches bereits in Abschnitt 2.3.4 erwähnte wurde. [BD10] Weitere Merkmale einer CEP Engine wird im Laufe der Arbeit mit den vorgestellten Engines Siddhi in Kapitel 3 und Esper in Abschnitt 2.5 genauer erläutert.

Event Processing Network

Event Processing Network beschreibt einen Verbund von EPAs. Die EPAs sind zwar vom Grundprinzip her identisch, können aber unterschiedliche Aufgaben übernehmen, d. h. um einen hohen Verarbeitungsaufwand eines einzelnen EPA zu vermeiden, werden komplexe Ereignisregeln in einfachere Teilregeln aufgespalten und auf mehrere EPAs verteilt. Daher könnte ein EPA beispielsweise Ereignisse auf bestimmte Eigenschaften filtern und ein anderer EPA könnte eine definierte Mustererkennung durchführen. Demzufolge müssen einzelne EPAs nur noch eine geringere Anzahl an Ereignisregeln ausführen. Die Kommunikation der EPAs untereinander erfolgt über das Netzwerk, anhand von Ereigniskanälen, sodass innerhalb der Verarbeitungsschritte zwischen den EPAs die Ereignisse ausgetauscht werden. [BD10]

2.5. CEP Engine „Esper“

Dieser Abschnitt befasst sich mit der Vergleichsengine Esper. Dabei sollen kurz auf die wesentlichen Punkte eingegangen werden.

2.5.1. Überblick

Esper ist eine in Java geschriebene, frei verfügbare CEP Engine, die unter der GNU General Public License (GPL) steht, und wurde im Jahr 2006 vom Unternehmen EsperTech veröffentlicht. [Esp18h] Esper unterscheidet sich zwischen zwei Varianten: Einmal Esper für Java und die Variante NEsper für das .NET Framework (Microsoft .NET). Des Weiteren verfügt Esper für Geschäftspartner die kommerzielle Esper Enterprise Edition. [Esp18e] Für die Umsetzung von Ereignisregeln bietet Esper die Esper Query Language (EQL), welche zu der Datenstrom-Anfragesprachen CQL angehört. [Esp18f] Diese Art von Ereignisanfragesprachen wurde bereits in Abschnitt 2.3.5 erwähnt.

Ein erster Einblick in die EQL bietet die folgende Ereignisregel eines Ereignismusters anhand des vorgestellten Temperaturüberwachungsszenarios:

```
1 SELECT t.temp
2 FROM pattern [wz = WindowEvent(f="zu") -> NOT wa = WindowEvent
     (f="auf", wz.roomNu = roomNu) AND
3 t = TempEvent(temp < 10, wz.roomNu = roomNu)]
```

Auflistung 2.1: Ereignisregel eines Ereignismuster in der EQL

Wie in Code-Beispiel 2.1 zu sehen, enthält die EQL die notwendige `SELECT`- und `FROM`-Klausel, welche jeweils an der SQL-Standardsprache angelehnt sind und zur Ereignis- und Zeitauswertung dienen. Die `SELECT`-Klausel gibt die Ereigniseigenschaften an, die betrachtet werden sollen. Die `FROM`-Klausel gibt den Ereignistyp an, der die Ereignisse aus einem bestimmten Ereignisstrom spezifiziert, um auf diesem die EQL-Anfrage anzuwenden. Je nach Definition der Ereignisregeln können auch weitere Klauseln wie z. B. die `WHERE`-Klausel genutzt werden, die eine Bedingung zur Suche von einem Ereignis oder von Ereigniskombinationen angibt. Sobald eintreffende Ereignisse den Bedingungen der EQL-Anfrage entsprechen, werden diese an den Listener bzw. Subscriber veröffentlicht. [Esp18f]

Die Ereignisregel in Code-Beispiel 2.1 stellt eine komplexere Ereignisanfrage dar und setzt das aus Abschnitt 2.3.3 bereits besprochene Muster in EQL um. Dafür wird die EQL-spezifische `PATTERN`-Klausel mit jeweils einem Sequenz-, Konjunktions-, Negations- und Vergleichsoperator genutzt. Genaueres zu den Sprachelementen und Operatoren der EQL werden später in Abschnitt 4.4.2 anhand der Ereignisregeln der Fallstudie erklärt.

In einer Esper-Anwendung werden Ereignisreaktionen und Ereignistypen standardmäßig in Java realisiert. So werden die Ereignisse in Esper intern als POJOs (Plain Old Java Objects), Java Beans, Maps, Objekt-Arrays oder auch in XML verarbeitet. Dennoch gibt es die Möglichkeit, auch die Deklaration eines Ereignistyps über die Klausel `create schema` vorzunehmen. [Esp18f]

Diese Arbeit stützt sich auf die letzte stabile Esper Version 7.1.0[2] vom 9. März 2018 und wird dementsprechend für die Umsetzung der Fallstudie genutzt.

[2] Aktuelle und ältere Esper Versionen finden sich unter http://esper.espertech.com/distributions/.

3. Einführung in die CEP Engine „Siddhi“

Dieses Kapitel gibt einen kurzen Überblick auf die CEP Engine Siddhi, den Aufbau und die Formulierung einer Siddhi-Anwendung und konzentriert sich auf ihre Basisarchitektur sowie Ablauf und Zusammenspiel der Komponenten innerhalb einer Siddhi-Anwendung.

3.1. Überblick

Siddhi ist eine leichtgewichtige, skalierbare und frei verfügbare CEP Engine und wurde unter der Apache Software License v2.0 als Java Bibliothek veröffentlicht. Zusammen mit der WSO2 Inc. und der Universität Moratuwa in Sri Lanka wurde Siddhi als ein Forschungsprojekt begonnen und entwickelt. [WSO18d] Inzwischen wird die CEP Engine nur noch von WSO2 weiter unterstützt und bildet die Kernengine des WSO2 Complex Event Processors (WSO2 CEP). Dementsprechend steht Siddhi eingebettet im WSO2 CEP[1] oder eingebunden als Bibliothek in Java zur Verfügung. [WSO18d, WSO18m]

Nach Angaben des Unternehmens Uber[2], welches Siddhi zur Betrugsanalyse einsetzt, zeigt sich die CEP Engine durchaus leistungsstark, da sie 200 Milliarden Ereignisse pro Tag verarbeitet, d. h. im Durchschnitt werden 300 tausend Ereignisse pro Sekunde bewältigt. Zudem ist Siddhi äußerst leichtgewichtig und kann daher in Android[3] oder auf einem RaspberryPi integriert werden. [WSO18c]

Jegliche Anforderungen von CEP wie Berechnung von Aggregationen (siehe Abschnitt 2.3.3) über verschiedene Fensterarten (siehe Abschnitt 2.3.4), Zusammenführen von mehreren Ereignisströmen, Interaktionen mit Datenbanken zur Kontextanreicherung, Datenkorrelation und Erkennung von zeitlichen Ereignismustern (siehe Abschnitt 2.3.3) können mit Siddhis Ereignisregeln umgesetzt werden. [WSO18c]

[1] Weitere Informationen über den WSO2 CEP, der ebenfalls unter der Apache Software License v2.0 steht, sind unter [WSO18m] zu finden.
[2] Uber ist ein Dienstleistungsunternehmen, das Online-Vermittlungsdienste zur Personenbeförderung anbietet.
[3] Siddhi Android Platform - https://wso2.github.io/siddhi-android-platform/

In Abschnitt 2.3.5 wurden die verschiedenen Arten der Ereignisanfragesprachen erwähnt. Siddhi nutzt zur Definition von Ereignisregeln und Ereignisströmen seine Siddhi Streaming SQL, die der Datenstrom-Anfragesprache CQL angehört und somit auf der deklarativen Sprache SQL beruht. [WSO18a]

Weitere konzeptionelle und sprachliche Einblicke der Siddhi Streaming SQL erfolgt in Abschnitt 4.3 und Kapitel 5. Diese Arbeit bezieht sich dabei auf die aktuelle Siddhi Entwicklungsversion 4.2.31 vom 02. November 2018[4].

3.2. Architektur

Siddhis Architektur besteht aus mehreren *Prozessoren* und repräsentieren die EPAs einer EDA, die bereits in Abschnitt 2.4.2 erklärt wurden. Diese Prozessoren sind durch mehrere *Warteschlangen* (queues) miteinander verbunden und dienen zum internen Ereignistransport. Zur Kommunikation verwenden die Prozessoren als Kommunikationsmechanismus das Publish&Subscribe-Prinzip, sodass sich nachfolgende Prozessoren bei vorgeschalteten Prozessoren registrieren können, um deren erzeugten Ereignisse zu empfangen. [WSO18d]

Die eintreffenden Ereignisse werden über einen *Input Adapter* aus verschiedenen Quellen konsumiert und bilden intern eine Datenstruktur in Form von Tupeln, d. h. infolge eines beispielsweise eintreffenden POJOs über den Input Handler, konvertiert Siddhi das Ereignis in ein Tupel. [SGN+le] Dadurch wird der Zugriff auf die Daten eines Ereignis wesentlich effizienter im Vergleich zu anderen Datenstrukturen wie z. B. XML. [SGN+le]

Die konsumierten Ereignisse werden anschließend in den Warteschlangen platziert und von mehreren Prozessoren „abgehört“ und verarbeitet. Die Prozessoren unterscheiden sich jedoch untereinander und führen jeweils andere Berechnungen durch. Jeder Prozessor enthält dafür sogenannte *Executoren*. [SGN+le] Ein Executor enthält die Verarbeitungslogik für die eintreffenden Ereignisse und stellt somit die, in Abschnitt 2.4.2 eingeführte Komponente, CEP Engine dar.

Jeder Executor realisiert eine definierte Bedingung (aus Kernoperatoren, siehe Abschnitt 2.3.3) einer Ereignisabfrage, d. h. ein Executor verarbeitet die eintreffenden Ereignisse und benachrichtigt, ob ein Ereignis die Bedingung erfüllt. Trifft das Ereignis auf die Bedingung zu, wird es zur weiteren Verarbeitung an den nächsten Executor weitergeleitet. Alle nicht zutreffenden Ereignisse werden verworfen. [SGN+le]

Nach vollständiger Verarbeitung der Ereignisse innerhalb eines Prozessors werden diese in Ausgabewarteschlangen platziert. Andere Prozessoren können Ereignisse aus Ausgabewarteschlangen weiterverarbeiten oder werden dem Benutzer als Ereignisbenachrichtigung gesendet. Des Weiteren können Ausgabeereignisse über die *Output Adapter* an

[4] Alle Versionen von Siddhi können unter http://maven.wso2.org/nexus/content/groups/wso2-public/org/wso2/siddhi/siddhi/ gefunden werden.

abonnierte Ereignissenken veröffentlicht werden, z. B. an externe Systeme zur anderweitigen Verarbeitung. [SGN+le, WSO18d]

Ein Prozessor kann durchaus mit mehreren Ereignisströmen verbunden sein. Auf dieser Basis verwendet Siddhi eine einzige Eingangswarteschlange und sammelt alle Ereignisse zusammen. Dadurch vereinfacht sich die Prüfung, ob Ereignisse aus allen Eingangswarteschlangen schon verarbeitet wurden oder nicht. Trotz Vermischung der Ereignisse kann ein Prozessor ein Ereignis seinem ursprünglichen Ereignisstrom zuordnen. Die Ereignisse werden mit IDs ihres abstammenden Ereignisstroms versehen und können so präzise verarbeitet werden. [SGN+le, WSO18d]

Die folgende Abbildung 3.1 zeigt die in Siddhi Core befindlichen Ereigniswarteschlangen und Prozessoren. [WSO18d]

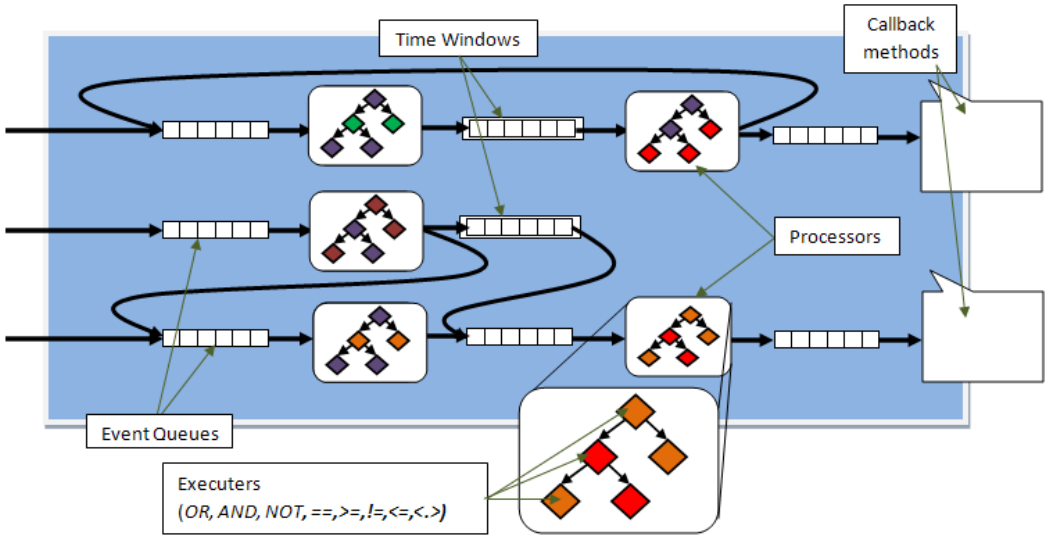


Abbildung 3.1.: Siddhi Core mit den enthaltenen Ereigniswarteschlangen und Prozessoren [SGN+le]

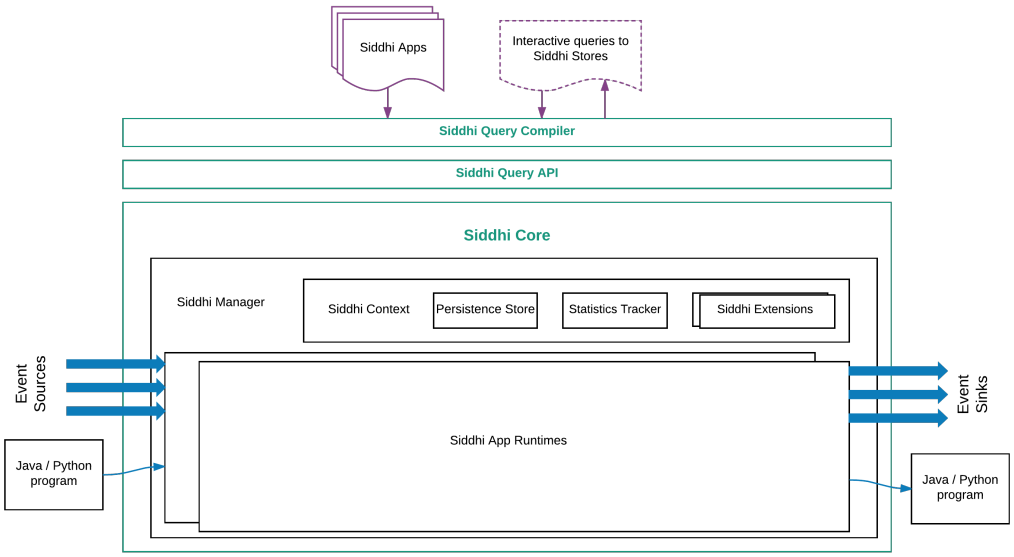


Abbildung 3.2.: Siddhi Architektur mit den enthaltenen Hauptmodulen und Komponenten [WSO18d]

Die Abbildung 3.2 zeigt die Basisarchitektur von Siddhi mit den enthaltenen Hauptmodulen (grün umrandet) und Komponenten, die im Folgenden erklärt werden.

- **Siddhi Query Compiler:** Konvertiert die Siddhi-Anwendung in Prozessoren, welche aus definierten Ereignisströmen und Ereignisanfragen besteht. [WSO18d]
- **Siddhi Core:** Verwaltet die Ausführungslogik und baut die Laufzeitumgebung der definierten Siddhi-Anwendung zusammen. Stellt die Prozessoren bereit und greift zur Verarbeitung von Ereignissen auf Warteschlangen zu. Interagiert auch mit externen Systemen, um Ereignisse zu konsumieren und zu veröffentlichen. [WSO18d]
- **Siddhi Query API:** Kann Ereignisanfragen und Ereignisströme mithilfe ihrer bereitgestellten Methoden der POJO-Klassen definieren. [WSO18d]
- **Siddhi Query Annotation:** Mit diesem Modul lassen sich mit speziellen Annotationen die Erweiterungen (z. B. Datenextraktion, Ereignispersistenz in Datenbanken etc.) von Siddhi benutzen, die zur Laufzeit vom Siddhi Core selektiert werden. [WSO18d]

Folgende Komponenten werden vom Siddhi Core verwendet:

- Der **Siddhi Manager** verwaltet, wie in Abbildung 3.2 erkenntlich, die beiden Komponenten *Siddhi Application Runtime* und *Siddhi Context*. [WSO18d]

- Eine **Siddhi Application Runtime** wird als Laufzeitobjekt für jede Siddhi-Anwendung erzeugt und bietet eine isolierte Laufzeitumgebung. Sie ist beispielsweise für das Konsumieren und Weiterleiten der Ereignisse zwischen weiteren Siddhi-Anwendungen, in Java-Programmen oder Python-Programmen zuständig, die ebenfalls Siddhi-EPAs enthalten. [WSO18d]
- Der **Siddhi Context** ist ein gemeinsames Objekt, welches für alle Siddhi Application Runtimes innerhalb des Siddhi Managers genutzt wird. Es enthält Referenzen auf den *Persistence Store* für periodische Persistenz z. B. für Snapshots der Verarbeitungszustände, auf den *Statistic Tracker*, um Leistungsstatistiken während der Laufzeit einer Siddhi Anwendung auszuwerten und mitzuteilen, und auf sogenannte *Extension holders* zum Laden von Siddhi Erweiterungen. [WSO18d]

3.3. Aufbau und Formulierung einer Siddhi-Anwendung

Anders als bei den meisten CQL-basierten CEP Engines, werden in Siddhi beispielsweise keine POJO-Klassen oder XML-Dokumente benutzt, um die Ereignistypen zu definieren. Stattdessen nutzt Siddhi nur die Definition eines Ereignisstrom, der implizit den Ereignistyp der eintreffenden Ereignisse festlegt. Die eintreffenden Ereignisse bekommen entsprechend eine Ereignisstrom-ID, die den Ursprungsstrom eines Ereignisses festhält. Des Weiteren enthalten Ereignisse einen Zeitstempel, gemäß der Systemzeit auf dem die Engine läuft und ein Objekt-Array für die spezifischen Attribute des Ereignisses. Damit eintreffende Ereignisse einem Ereignisstrom zugeordnet werden, wird dieser zuerst in einer Siddhi-Anwendung definiert. Ein Ereignisstrom in Siddhi wird mit `define stream` folgendermaßen realisiert:

```
define stream [Name des Ereignisstroms](Attribut 1, Attribut 2, Attribut 3)
```

Dieser Ausdruck spezifiziert den Ereignisstrom und beinhaltet den Namen des Ereignisstroms sowie die Attribute mit ihren primitiven Datentypen.

Wie bereits erwähnt, basieren die Siddhi Ereignisanfragen auf der deklarativen Datenbankabfragesprache SQL und werden mithilfe der Anweisungen wie z. B. `SELECT, FROM, JOIN, GROUP BY` usw. erstellt. Eine Siddhi Ereignisregel besitzt dabei folgende Struktur:

```
SELECT <Attribute des Ausgangsstroms>
FROM <Eingangsstrom>[<Filterbedingung>]#<Windows, z. B. Zeitfenster>
INSERT INTO <Ausgangsstrom>
```

Durch einen festgelegten Filter, fließt nur eine bestimmte Menge an eingehende Ereignisse in das angegebene Window. Je nach ausgewählter Fensterart (z. B. Zeitfenster, siehe Abschnitt 2.3.4) erhält dieses auf bestimmte Zeit die Ereignisse, die für die weitere

Verarbeitung in Frage kommen. Anschließend werden die resultierenden Ereignisse in die Ausgangsströme übertragen.

Sind schließlich ein oder mehrere Ereignisströme und Ereignisanfragen definiert, so wird die Deklaration an den Siddhi Manager übergeben, um anschließend das Laufzeitobjekt vom Typ Siddhi Application Runtime zu erstellen und auszuführen.

Folgendes Code-Beispiel 3.1 zeigt die Definition eines `TempEventStreams` und eines `WindowEvenStreams`. Die beiden Ereignisströme legen jeweils ein Ereignisschema fest. Die eintreffenden Ereignisse eines Ereignistyps, z. B. `TempEvent`, werden genau ihrem Ereignisstrom `TempEventStream` zugeordnet, welches das gleiche Ereignisschema besitzt, d. h. Ereignisse müssen die gleichen Attribute in der selben Reihenfolge besitzen wie das Ereignisschema ihres entsprechenden Ereignisstroms. Das weitere Code-Beispiel 3.2 zeigt die Umsetzung einer Ereignisregel mit dem Ereignismuster aus Abschnitt 2.3.3. Die beiden Code-Beispiele sind ebenfalls auf das Szenario der Temperaturüberwachung aus Abschnitt 2.2 bezogen.

```
1 define stream TempEventStream (sensorID string, roomNu int, temp
     double);
2 define stream WindowEventStream (sensorID string, roomNu int, f
     string);
```

Auflistung 3.1: Definition zweier Ereignisströme mit der Siddhi Streaming SQL

```
1 from every (wz = WindowEventStream[f == 'zu']) -> t =
     TempEventStream[temp < 10 and wz.roomNu == roomNu] and not
     wa = WindowEventStream[f == 'auf' and wz.roomNu == roomNu]
2 select  t.temp
3 insert into AlertStream;
```

Auflistung 3.2: Definition der Ereignisregel eines Ereignismusters mit der Siddhi Streaming SQL

3.4. Ereignisverarbeitung innerhalb der Siddhi-Anwendungslaufzeit

In diesem Abschnitt soll auf das Zusammenspiel der Komponenten innerhalb der Siddhi-Anwendungslaufzeit eingegangen werden.

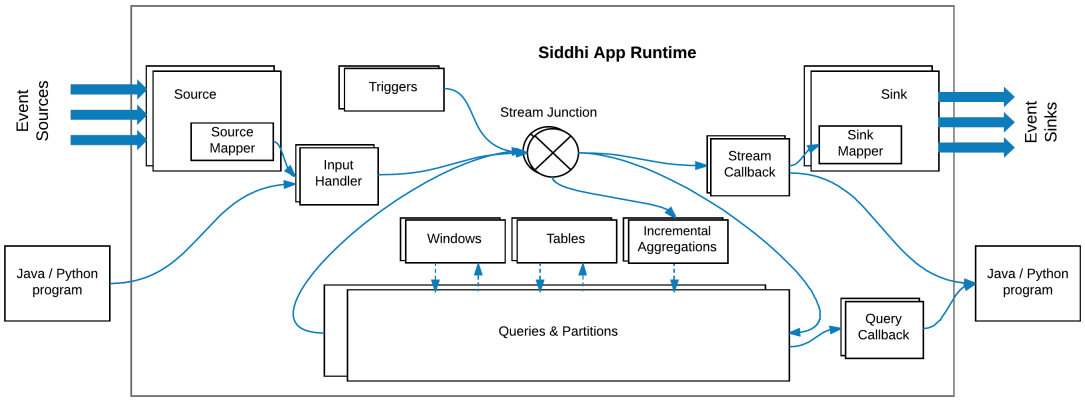


Abbildung 3.3.: Der Ablauf und das Zusammenspiel der Komponenten innerhalb der Siddhi Anwendungslaufzeit [WSO18d]

In Siddhi durchlaufen Ereignisse verschiedene Komponenten innerhalb der Siddhi-Anwendungslaufzeit. Die Abbildung 3.3 zeigt den Ablauf anhand der blauen Linien.

Zuerst fließen die Ereignisse, die in verschiedenen Datenformaten aus externen Ereignisquellen auftreten, in die *Source-Komponente*. Die Source-Komponente kann hierbei als Input Adapter angesehen werden. Für jeden definierten Ereignisstrom wird mit der `@Source`-Annotation eine Source-Komponente generiert. Damit die eingehenden Ereignisse weiterverarbeitet werden können, muss ein *Source Mapper* der Source-Komponente diese Ereignisse in das richtige Siddhi-Ereignisformat konviertien, welches am Anfang dieses Abschnitts bereits besprochen wurde. Mithilfe der `@Map`-Annotation innerhalb einer `@Source`-Annotation kann ein Ereignismapping vorgenommen werden. Wird jedoch keine `@Map`-Annotation angegeben, bleiben jegliche Änderungen aus. Siddhi nimmt an, dass die Ereignisse im richtigen Ereignisformat vorliegen und nutzt dabei den Source Mapper vom Typ `PassThrough`. [WSO18d]

Ereignisse werden ggf. nach einer Transformation von der Source-Komponente zum *Input Handler* weitergeleitet. Der Input Handler ist dafür verantwortlich, die Ereignisobjekte der Source-Komponente oder aus Java-Programmen zur entsprechenden *Stream Junction* zu schicken. Eine Stream Junction wird für jeden Ereignisstrom generiert und erzeugt eine eigene InputHandler-Instanz. Diese Stream Junction übernimmt anschließend die

Weiterleitung der Ereignisse an die Komponenten innerhalb der Laufzeit, die für den Ereignisstrom registriert sind. [WSO18d]

Mit speziellen `@Async`-Annotation kann dazu beigetragen werden, dass die Stream Junction die Ereignisse puffert und in weitere Threads lagert, um für die Ereignisströme asynchrones Verhalten festzulegen (weiteres in Abschnitt 5.1.7). [WSO18a] Zur Filterung, Aggregation, Transformation oder weitere Verarbeitungen gelangen die Ereignisse, wie in Abbildung 3.3 erkenntlich, aus ein oder mehreren Stream Junctions in die Prozessoren. Hier werden nur die Ereignisse ausgegeben und in Siddhi Windows eingespeist, die der Filterbedingung entsprechen. Des Weiteren arbeitet die Komponente auch mit Ereignistabellen, die zur Ereignispersistenz verwendet werden. Siddhi speichert standardmäßig die Ereignistabellen im Arbeitsspeicher. [WSO18d]

Sobald die Ereignisse verarbeitet wurden, gelangen diese an ein Callback. Mit dem *Query Callback* werden die Ereignisse abhängig nach der definierten Ereignisanfrage mit Zeitstempel ausgegeben und zwischen aktuelle und abgelaufene Ereignisse klassifiziert. [WSO18d] Der *Stream Callback* hingegen ist an einem Ereignisstrom registriert und sendet die Ereignisse an die *Sink-Komponente* sowie an weitere abonnierte Siddhi-Anwendungen eines Java-Programms. Die Sink-Komponente wird mit `@Sink`-Annotation für jeden Ereignisstrom generiert, sodass alle Ereignisse in den Ausgangsstrom gelangen und an externe Endpunkte veröffentlicht werden. In Siddhi gibt es auch einen implizit definierten Ausgangsstrom (`OutputStream`), welcher oft zur Ereignisbenachrichtigung für den Benutzer verwendet wird. Sollen die Ereignisse in ein bestimmtes Datenformat publiziert werden, kann dies wieder mit der `@Map`-Annotation geschehen. Der bereits erwähnte Sink Mapper konvertiert daraufhin die Ereignisse in das festgelegte Datenformat. Liegt jedoch keine Konfiguration mit der Annotation vor, wird das standardmäßige Siddhi-Ereignisformat genutzt. Das Siddhi-Ereignisformat ist das interne *StreamEvent* und wird erstellt, bevor es an die Ereignisregeln gesendet wird. Es besitzt den Zeitstempel und die spezifizierenden Ereignisattribute. Die Ereignisattribute sind dabei in mehreren Java Object-Arrays enthalten, um zwischen wichtigen Ausgangsattributen oder Attributen, die für die Verarbeitung während oder nach einem Window relevant sind, zu unterscheiden. [WSO18d]

3.5. Ablauf innerhalb der Siddhi-Ereignisanfragen

In Abschnitt 3.4 wurden die Komponenten und der Ablauf innerhalb der Siddhi-Anwendungslaufzeit erklärt. In diesem Abschnitt soll auf den Ablauf innerhalb der Siddhi-Ereignisanfragen genauer eingegangen werden. Für jede Filter-, Muster-, Join- oder Sequenzanfrage wird in Siddhi ein entsprechender Typ eines *Input Streams* erzeugt. Im Folgenden wird aber nur auf dem *Single Input Stream* für Filteranfragen eingegangen.

3.5.1. Single Input Stream

Ein Single Input Stream wird für jede einzelne Ereignisanfrage, die aus Filtern oder Windows bestehen kann, erzeugt. Alle eingehenden Ereignisse aus einer Stream Junction oder aus einem Window werden im Single Input Stream eingespeist und basierend auf das Ausgabeformat des Ausgabeereignisstroms umgewandelt, d. h. alle Ereignisstromattribute werden gelöscht, die nicht weiter in Betracht gezogen werden. [WSO18d]

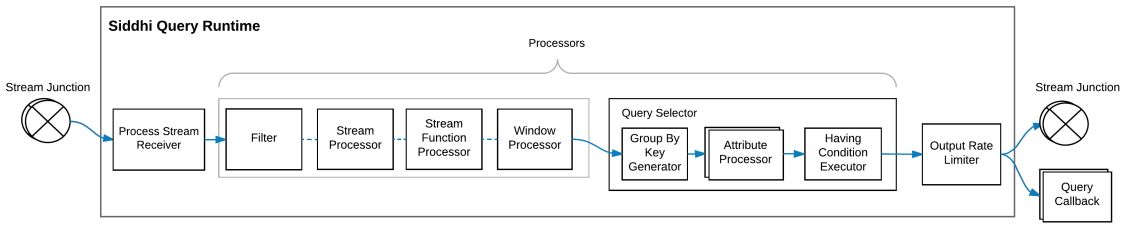


Abbildung 3.4.: Pipeline eines Single Input Streams in Siddhi [WSO18d]

Wie in Abbildung 3.4 zu sehen gelangen die konvertieren Ereignisse durch eine Prozessor-Pipeline, bestehend aus dem *Filter Processor*, *Stream Processor*, *Stream Function Processor*, *Window Processor* und dem *Query Selector*. Am Anfang von Abschnitt 3.2 wurden die Executoren in Siddhi eingeführt. Der Filter Processor ist mit einem Executer, dem sogenannten *Expression Executer*, ausgestattet und gibt bei Auswertung einer definierten Bedingung einen booleschen Wert zurück, ob die Bedingung erfüllt ist oder nicht. [WSO18d]

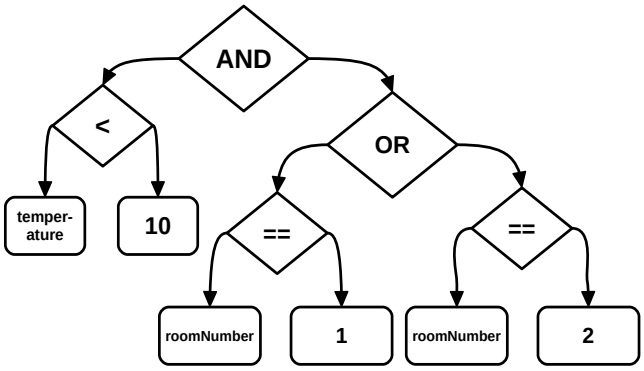


Abbildung 3.5.: Baumstruktur von mehreren Filterbedingungen [vgl. WSO18d]

Die Abbildung 3.5 stellt die Executoren bzw. die Ausdrücke einer Bedingung als Baumstruktur dar, die nach dem Tiefensuchalgorithmus verarbeitet werden. Ein Ereignis durchläuft ab der Wurzel beginnend, die Baumstruktur. Erfüllt das Ereignis alle Bedingungen des Baumes wird `true` zurückgeliefert, ansonsten `false`. Um in Siddhi die

Verarbeitung der Bedingungen möglichst effizient zu gestalten, ist es sinnvoll den unwahrscheinlichsten Fall, der eintritt, auf der linken äußeren Seite einer Bedingung zu definieren. Aufgrund dessen lassen sich Ereignisse schneller herausfiltern, wenn diese bei der Bedingung fehlschlagen. [WSO18d]

Die Prozessor-Pipeline endet immer mit dem Query Selector. Im Query Selector trifft jedes von den vorherigen Prozessoren verarbeitete Ereignis ein und wird auf Basis der `SELECT`-Klausel umgewandelt. Zum Beispiel wird mit der `GROUPBY`-Klausel ein spezieller Gruppenschlüssel generiert, sodass der Query Selector die gruppierte Ereignismenge identifizieren kann. Dadurch lässt sich im weiteren Verlauf der Attribute Processor auf diese Ereignismengen anwenden. Die Attribute Processors enthalten dabei die Ausdrücke mit konstanten Werten, Variablen und ggf. Attribute Aggregators, um Aggregationsoperatoren wie `Avg(), Sum(), Min()` oder `Max()` zu verwenden. Nach Ausführung der Aggregationsfunktionen werden die Ereignisse wieder in das entsprechende Ausgabeformat konvertiert und mit dem *Having Condition Executer* nach der `HAVING`-Klausel ausgewertet, falls diese in der Ereignisanfrage enthalten ist. [WSO18d]

Ist eine `OUTPUT`-Klausel in der Ereignisanfrage enthalten, wird zum Schluss eine Ereignisausgabe über den *Output Rate Limiter* zur Stream Junction oder dem Query Callback geschickt. Der Output Rate Limiter sorgt dafür, dass Ausgaben begrenzt werden. Dadurch werden nachfolgende Ausführungen nicht belastet und unwichtige Informationen entfernt. Erst danach werden die resultierenden Ereignisse weitergeleitet. Sollte jedoch keine `OUTPUT`-Klausel vorliegen, werden die Ereignisse ohne weitere Einschränkung übergeben. [WSO18a, WSO18d]

3.5.2. Ereignisverarbeitung mit Windows

Wie bereits in vorherigen Abschnitten erwähnt, wird die zeitliche Verarbeitung von Ereignissen in Siddhi mithilfe von Windows umgesetzt. Dabei klassifiziert Siddhi die Ereignisse unterschiedlich [WSO18d]:

- Die **Current Events** definieren die zur Verarbeitung neu eintreffenden Ereignisse.
- Die **Expired Events** stellen die im Window abgelaufenen Ereignisse dar.
- Die **Timer Events** werden vom Scheduler erzeugt. Sie geben Auskunft, wann ein Ereignis abgelaufen ist.

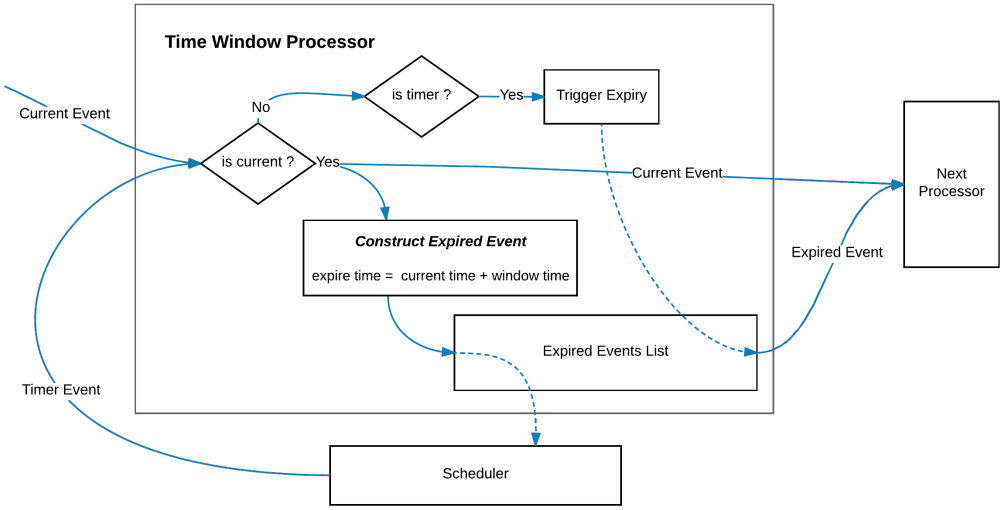


Abbildung 3.6.: Zeitliche Verarbeitung innerhalb eines Windows [WSO18d]

Die Abbildung 3.6 zeigt die zeitliche Verarbeitung innerhalb eines Window Processors, indem die Ereignisse folgendermaßen behandelt werden:

Für jedes aktuelle Ereignis (current event) erzeugt der Window Processor ein abgelaufenes Ereignis (expired event) mit einem ablaufendem Zeitstempel (akutelle Zeit + Fensterzeit) und speichert dieses im aktuellen Window. Das aktuelle Ereignis wird zusätzlich vom Window Processor zur weiteren Verarbeitung an den nächsten Prozessor geschickt, z. B. an den Query Selector. [WSO18d]

Für das abgelaufene Ereignis wird ein Eintrag in eine Liste des *Scheduler* (Planer) hinzugefügt. Die Liste enthält dabei die Einträge aller abgelaufenen Ereignisse, die sich im Speicher befinden. Der Scheduler sendet, sobald ein Ereignis im Window abgelaufen ist, eine Benachrichtigung als Timer Event. Das Window erhält anschließend das Timer Event und schickt das älteste abgelaufene Ereignis an den nächsten Prozessor und löscht es aus dem Window. [WSO18d]

In Siddhi ist der Verarbeitungsablauf eines Windows aus Abbildung 3.6 besonders relevant, da bei einer Aggregation im Query Selector nach aktuellen und abgelaufenen Ereignissen unterschieden werden muss. Sollen bzw. alle Ereignisse (current events und expired events) ausgegeben werden, wird ein sogenanntes *Reset Event* gesendet, welches den Zustand der Aggregationsberechnung zurücksetzt. Sollen nur die aktuellen Ereignisse ausgegeben werden, muss für jedes aktuelle Ereignis ein abgelaufenes Ereignis gesendet werden. Dies hat die Absicht, dass aktuelle Ereignisse die Werte der Aggregationsberechnung erhöhen und abgelaufene Ereignisse diese Werte verringern. Das Reset Event setzt die Aggregationsberechnung entsprechend zurück. [WSO18d]

4. Fallstudie

In diesem Kapitel werden die Fallstudie und die zu implementierenden Ereignisregeln vorgestellt. Anschließend wird die Umsetzung der beiden Vergleichskandidaten Siddhi und Esper mit der Fallstudie jeweils separat betrachtet und resultierende Ergebnisse diskutiert.

4.1. Modellbeschreibung der Fallstudie

Die Analyse und Verarbeitung von Gesundheitsdaten in Echtzeit ist mittlerweile unentbehrlich und ein wichtiger Aspekt im Gesundheitswesen. Der Einsatz von Gesundheitsanwendungen in Krankenhäusern erfordert einige Verarbeitungs- und Netzwerkanforderungen, die oft durch die begrenzte Infrastruktur nicht erfüllt werden können. Außerdem müssen die Anwendungen mit hochfrequenten Sensordaten aus unterschiedlichen Quellen (z. B. Biosensoren zur Blutdruckmessung) umgehen können und ein zuverlässiges, skalierbares und ausfallsicheres System erbringen. [YCL11] Der Einsatz einer EDA mit der Softwaretechnologie CEP kann daher eine passende Lösung sein, um diese Anforderungen zu bewältigen. In dieser Arbeit soll als Fallstudie eine selbst erdachte *Gesundheitsüberwachung* simuliert werden. Die Wahl dieser Fallstudie, soll zeigen wie wichtig der Einsatz von CEP im Gesundheitsbereich ist. Oftmals sind Sekunden entscheidend die kleinsten Anomalien bei herzkranken Patienten zu erkennen und entsprechend zu reagieren. Vor allem bei Patienten, die sich beispielsweise nicht in Krankenhäusern oder Pflegeeinrichtungen befinden, können die Gesundheitsdaten per Ferngesundheitsüberwachung erfasst und analysiert werden. So kann in Notfällen, wenn sich ein Patient in einer kritischen gesundheitlichen Situation befindet, ein Arzt kontaktiert werden.

Die umzusetzende Fallstudie Gesundheitsüberwachung mit den beiden CEP Engines soll zwei Vitalwerte von herzkranken Patienten einbeziehen, d. h. die Blutdruck- und Pulsfrequenzwerte werden gemessen und anschließend durch Ereignisanfragen ausgewertet. Das konstruierte Modell lässt sich folgendermaßen darstellen:

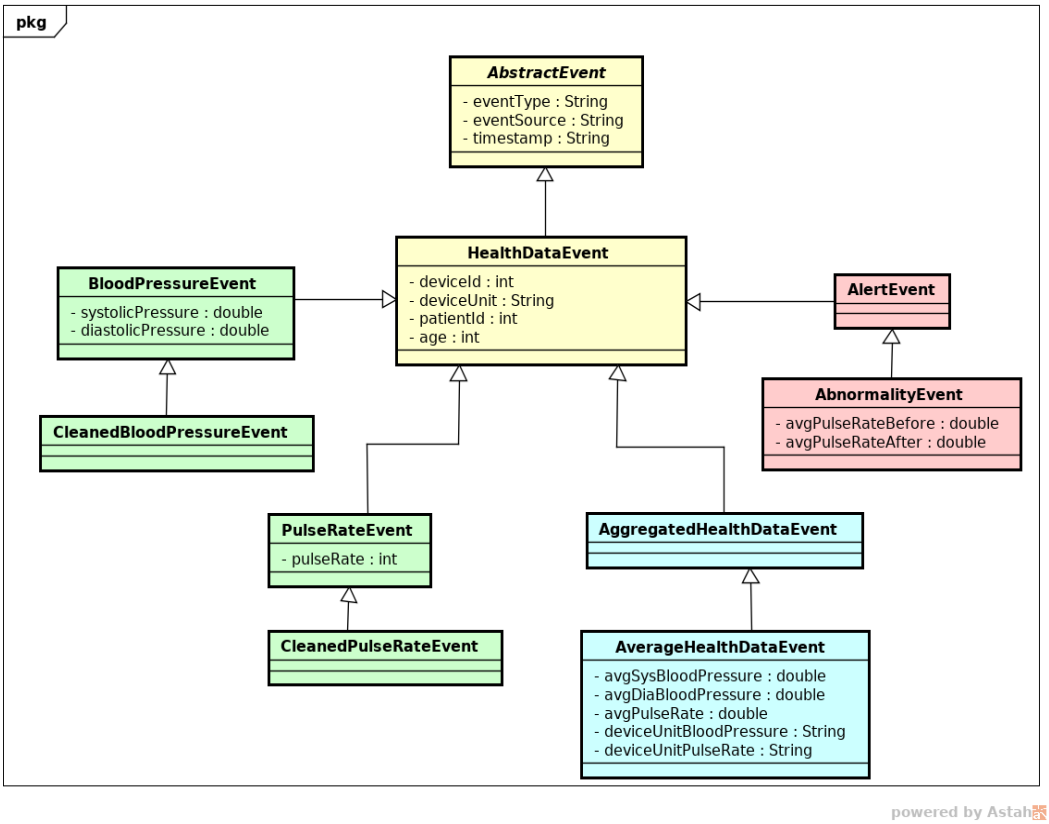


Abbildung 4.1.: Ereignismodell der Fallstudie *Gesundheitsüberwachung*

Vorweg erwähnt soll die Fallstudie als Mittel zur Evaluierung der beiden CEP Engines dienen, daher ist die medizinisch sinnvolle Auswertung der Vitalwerte absichtlich einfach gehalten.

4.1.1. Ereignisse

Folgende Ereignisse für die Fallstudie werden im Kurzen erklärt und in die jeweiligen Abstraktionsebenen aus Abschnitt 2.3.1 eingestuft:

- **`BloodPressureEvent (Blutdruckereignis):`** Ein atomares Ereignis, das eine Blutdruckmessung eines Patienten repräsentiert.
- **`PulseRateEvent (Pulsfrequenzereignis):`** Ein atomares Ereignis, das eine Pulsmessung eines Patienten repräsentiert.
- **`CleanedBloodPressureEvent (Sauberes Blutdruckereignis):`** Steht für ein gefiltertes Ereignis aus den `BloodPressureEvents` und enthält keine fehlerhaften Daten.

- **`CleanedPulseRateEvent` (Sauberes Pulsfrequenzereignis):** Steht für ein gefiltertes Ereignis aus den `PulseRateEvents` und enthält keine fehlerhaften Daten.
- **`AverageHealthDataEvent` (Ereignis mit den durchschnittlichen Gesundheitsdaten):** Repräsentiert ein mittleres Abstraktionsereignis, welches die Durchschnittswerte der `CleanedBloodPressureEvents` und `CleanedPulseRateEvents` eines Patienten enthält.
- **`AbnormalityEvent` (Anomalieereignis):** Stellt ein komplexes Abstraktionsereignis dar, welches eine Anomalie aus den `AverageHealthDataEvents` aufweist.

4.1.2. Testdaten

Die Testdaten der Fallstudie wurden im Dateiformat CSV (Comma-separated values) generiert. Die Abbildung 4.2 zeigt einen Teilausschnitt der Blutdrucktestdaten.

```
Blutdruckänderung,2018-10-17 08:18:12,6619002,6619002,26,114.74, 78.55,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 10:48:21,8789167,8789167,85, 67.91, 44.46,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 11:26:47,4454513,4454513,48,101.15, 70.28,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 14:09:30,3002859,3002859,77,138.34, 91.09,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 14:41:30,3534139,3534139,57, 64.20, 45.37,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 16:11:05,6619002,6619002,26,114.49, 79.01,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 17:10:29,8286024,8286024,13, 83.05, 54.90,mmHg,Blutdruckmessgerät
Blutdruckänderung,2018-10-17 18:04:02,9147653,9147653,42,150.61,124.03,mmHg,Blutdruckmessgerät
```

Abbildung 4.2.: Ausschnitt aus den Blutdrucktestdaten

Ein Datensatz enthält jeweils den Ereignistyp [`eventType`], den Zeitstempel [`timestamp`], die Geräteidentifikation [`deviceId`], die Patientenidentifikation [`patientId`], das Alter des Patienten [`age`], den systolischen Blutdruckwert [`systolicPressure`], den diastolischen Blutdruckwert [`diastolicPressure`], die Maßeinheit [`deviceUnit`] und die Ereignisquelle [`eventSource`].

```
Pulsänderung,2018-10-17 08:18:12,8173803,8173803,57, 52,/min,Pulsoximeter
Pulsänderung,2018-10-17 10:48:21,2410232,2410232,23, 68,/min,Pulsoximeter
Pulsänderung,2018-10-17 11:26:47,2693449,2693449,12, 72,/min,Pulsoximeter
Pulsänderung,2018-10-17 14:09:30,3805036,3805036,27,114,/min,Pulsoximeter
Pulsänderung,2018-10-17 14:41:30,5259087,5259087,50, 68,/min,Pulsoximeter
Pulsänderung,2018-10-17 16:11:05,1749073,1749073,42, 75,/min,Pulsoximeter
Pulsänderung,2018-10-17 17:10:29,4749154,4749154,59, 89,/min,Pulsoximeter
Pulsänderung,2018-10-17 18:04:02,8452466,8452466,58,115,/min,Pulsoximeter
```

Abbildung 4.3.: Ausschnitt aus den Pulstestdaten

Die Abbildung 4.3 stellt dementsprechend einen Teilausschnitt für die Pulstestdaten dar. Der einzige Unterschied zwischen den Pulstestdaten und den Blutdrucktestdaten ist, dass bei einem Pulsfrequenzereignis nur ein Vitalwert, die Pulsfrequenz [`pulseRate`], gemessen wird.

4.2. Beschreibung der zu implementierenden Ereignisregeln

Mit der vorgestellten Fallstudie sollen in Siddhi und Esper folgende Ereignisregeln realisiert werden:

1. Ereignisregel - Filtern von fehlerhaften Blutdruck- und Pulsfreqenzereignissen

In der Realität kann es vorkommen, dass Vitalwerte falsch gemessen werden und somit fehlerhaft sind, z. B. wenn durch ein defektes Messgerät der Pulswert 0 oder andere utopische Werte gemessen werden. Daher soll die erste Ereignisregel alle eintreffenden Blutdruck- und Pulsfrequenzereignisse verschiedener Patienten filtern. Die Regelform würde folgendermaßen interpretiert werden: Wenn ein `BloodPressureEvent` eintrifft, dass einen systolischen Blutdruckwert zwischen 80 und 280 und einen diastolischen Blutdruck zwischen 40 und 200 aufweist, dann wird aus ihm ein `CleanedBloodPressureEvent` erzeugt. Ebenfalls die gleiche Prozedur für die `PulseRateEvents`. Trifft ein `PulseRateEvent` mit einem Pulswert zwischen 40 und 130 ein, dann wird aus diesem Ereignis ein `CleanedPulseRateEvent` generiert.

2. Ereignisregel - Komposition von Blutdruck- und Pulsfrequenzereignissen und Berechnung des Durchschnittswertes der einzelnen Vitalwerte während eines Zeitfensters

Die zweite Ereignisregel soll aus den sauberen Blutdruck- und Pulsfrequenzereignissen den Durchschnitt der Vitalwerte für jeden Patienten berechnen, die innerhalb einer Sekunde eintreffen. Die Realisierung der Ereignisregel geschieht durch eine Komposition von eintreffenden `CleanedBloodPressureEvents` und `CleanedPulseRateEvents`. Die Regelform lässt sich dabei folgendermaßen interpretieren: Treffen `CleanedBloodPressureEvents` und `CleanedPulseRateEvents` innerhalb des Zeitfensters von einer Sekunde ein, dann wird auf den vereinten Ereignispaaren eine Aggregationsberechnung für jeden Patienten durchgeführt und ein `AverageHealthDataEvent` erzeugt.

Für diese Regel sei erwähnt, dass der Zeitraum des Windows sinnvollerweise klein gewählt wurde, um schnelle Tests in angemessener Zeit durchzuführen. In der Praxis werden natürlich größere Zeiträume definiert.

3. Ereignisregel - Mustererkennung, wenn sich der durchschnittliche Pulswert eines Patienten um einen festgelegten Wert im Vergleich zum davor gemessenen durchschnittlichen Pulswert erhöht

Die dritte Ereignisregel soll aus berechneten Durchschnittswerten eine Anomalie erkennen können. Die Anomalie wird erkannt, wenn sich der durchschnittliche Pulswert eines Patienten um einen festgelegten Wert im Vergleich zum davor gemessenen durchschnittlichen Pulswert erhöht. Die Regelform lässt sich dabei folgendermaßen interpretieren: Wenn ein `AverageHealthDataEvent` eintrifft, dessen Pulswert eines Patienten sich um 25 oder mehr Pulsschläge pro Minute im Vergleich zum vorhergehenden `AverageHealthDataEvent` erhöht, dann ist das Muster erfüllt und ein `AbnormalityEvent` wird erzeugt. Dazu sei erwähnt, dass die Ereignisse nicht unmittelbar aufeinander folgen müssen, um das Ereignismuster zu erkennen.

4. Ereignisregel - Einfache Filteranfrage für den Performanztest

Die vierte Ereignisregel dient für den Performanztest in Abschnitt 5.2.2 und soll Patienten nach festgelegten Grenzwerten des Alters filtern. Die Regelform wird dabei folgendermaßen interpretiert: Trifft ein `PulseRateEvent` ein, dessen Alter des Patienten zwischen 18 und 80 liegt, dann wird dieses Ereignis ausgegeben.

5. Ereignisregel - Aggregationsanfrage mit Batch-Längenfenster für den Performanztest

Die fünfte Ereignisregel wird ebenfalls für Performanzzwecke in Abschnitt 5.2.2 verwendet. Sie soll das höchste Alter unter fünf Patienten ermitteln. Die Regelform lässt sich dabei folgendermaßen interpretieren: Sobald fünf `PulseRateEvents` in das Batch-Längenfenster eintreffen, wird auf dieser Ereignismenge eine Aggregation durchgeführt. Die Aggregation ermittelt aus den fünf `PulseRateEvents` das höchste Alter eines Patienten und gibt die Patient-ID und das Alter aus. Sind in der Ereignismenge mehrere Ereignisse vorhanden, die das gleiche Maximalalter enthalten, wird immer das zuletzt eintreffende `PulseRateEvent` im Längenfenster selektiert. Die Prozedur wird für alle weiteren eintreffenden `PulseRateEvents` durchgeführt.

6. Ereignisregel - Musteranfrage für den Performanztest

Die letzte Regel für den Performanztest in Abschnitt 5.2.2 soll erkennen, wenn sich der Pulswert eines Patienten im Vergleich zum davor gemessenen Pulswert erhöht hat. Die Regelform wird dabei folgendermaßen interpretiert: Trifft ein `PulseRateEvent`, dessen Pulswert höher als der des vorhergehenden `PulseRateEvents` ist, dann ist das Muster erfüllt und die Patient-ID sowie die beiden Pulswerte des Patienten werden ausgegeben.

4.3. Realisierung der Fallstudie mit Siddhi

In diesem Abschnitt wird die Umsetzung der Fallstudie mit Siddhi vorgestellt. Für die Implementierung wurde die zur Zeit aktive Siddhi Entwicklungsversion 4.2.0 verwendet. Im angelegten Maven-Projekt (Maven Version 3.6.0) wurden die entsprechenden Abhängigkeiten für Siddhi in die Konfigurationsdatei pom.xml[1] eingebunden.

4.3.1. Implementation der Siddhi-Anwendung

In Siddhi wird jegliche Verarbeitungslogik mit der Siddhi Streaming SQL implementiert. Wie in Abschnitt 2.4.2 bereits erwähnt, lässt sich die Verarbeitungslogik auf mehrere EPAs aufteilen. Für die Siddhi-Anwendung wurden die Ereignisregeln ebenfalls auf mehrere Siddhi EPA-Instanzen aufgeteilt, die dann untereinander die Ereignisse weiterleiten. In Code-Beispiel 4.1 wird die Siddhi Streaming SQL in Java als String eingebettet und definiert zwei Ereignisströme (Eingangs- und Ausgangsstrom) und eine Ereignisregel.

```
1  ClassLoader classLoader = SiddhiQuerysTest.class.getClassLoader();
2  String rootPath = classLoader.getResource("files/pulse_testData.csv").getPath();
3
4  String inputStream = ""+
5      "@source(type= 'file', mode='line', tailing= 'false', "+
6      "file.uri= 'file:/" + rootPath + "', @map(type='csv'))"+
7      "define stream PulseRateEventStream (eventType String, timestamp String, deviceId int,
8      patientId int, age int, pulseRate int, deviceUnit String, eventSource String); ";
9
10 String outputStream = ""+
11     "@sink(type='inMemory', topic='CleanedPrEvents', "+ "@map(type='passThrough'))"+
12     "define stream CleanedPulseRateEventStream (eventType String, timestamp String,
13      deviceId int, patientId int, age int, pulseRate int, deviceUnit String, eventSource
14      String); ";
15
16 String query = ""+
17     "from PulseRateEventStream[pulseRate > 40 and pulseRate < 130] "+
18     "select * "+
19     "insert into CleanedPulseRateEventStream;";
```

Auflistung 4.1: Umsetzung der Ereignisströme und einer Ereignisregel mit der Siddhi Streaming SQL in Java

In Abschnitt 3.3 wurde erläutert, dass in Siddhi die Ereignistypen mit der Definition der Ereignisströme implizit festgelegt werden. Der `PulseRateEventStream` realisiert den Eingangsstrom und enthält das entsprechende Ereignisschema mit den Attributen für die eingehenden `PulseRateEvents` (Zeile 7-8). Um die Ereignisse zu generieren, werden diese in Siddhi mithilfe der `@Source-Annotation` aus einer CSV-Datei gelesen. In der `@Source`-Annotation werden dafür verschiedene Parameter als Konfiguration übergeben

[1] Die benötigten Softwareabhängigkeiten für Siddhi lassen sich unter https://wso2.github.io/siddhi/documentation/user-guide/ und http://maven.wso2.org/nexus/content/groups/wso2-public/org/wso2/siddhi/siddhi/ finden.

(Zeile 5-6). Damit die Ereignistestdaten aus einer Datei gelesen werden, wird der `type`-Parameter mit dem Quelltypen `file` definiert. Die weiteren Parameter sind optionale Parameter der Siddhi-Erweiterung *siddhi-io-file*. Der `mode`-Parameter sorgt mit `line` dafür, dass die Ereignistestdaten in der CSV-Datei zeilenweise gelesen werden, da jede Zeile eine Ereignisinstanz repräsentiert. Der `tailing`-Parameter legt fest, ob nach dem Einlesen der Datei die Verbindung aufrecht gehalten wird, falls weitere Datensätze während der Ausführung hinzugefügt werden. Da der Gebrauch von `tailing` nicht weiter notwendig ist und der Defaultwert `true` beträgt, wurde es explizit auf `false` gesetzt. Mit dem `file.uri`-Parameter wird der Pfad zur entsprechenden Datei übergeben. Die `@Map`-Annotation gibt zum Schluss mit dem `type`-Parameter an, in welchem Datenformat die Datei vorliegen.

Der `CleanedPulseRateEventStream` realisiert den Ausgangsstrom, um die verarbeiteten `PulseRateEvents` nach der Ereignisanfrage an den Eingangstrom der anderen EPA-Instanz weiterzuleiten. Besonders zu beachten ist, dass der Ausgangsstrom in derselben EPA-Instanz erstellt werden muss, in der auch die Ereignisanfrage definiert ist. Des Weiteren muss der Ausgangsstrom vor der Definition der Ereignisregel festgelegt werden. Um nun den Transport für die verarbeiteten `PulseRateEvents` zu realisieren, muss zunächst am `CleanedPulseRateEventStream` die `@Sink`-Annotation hinzugefügt werden. Innerhalb der `@Sink`-Annotation werden ebenfalls Konfigurationsparameter übergeben (Zeile 11). Der Parameter `type` legt dabei den *In-Memory-Transport* fest, d. h. Ausgangsstrom und Eingangsstrom (Publisher und Subscriber), die miteinander kommunizieren, liegen in der gleichen Java Virtual Machine (JVM) und müssen das gleiche Ereignisschema besitzen. Durch das festgelegte Thema `CleanedPrEvents` über den Parameter `topic` kann der Subscriber das Thema des Publishers abonnieren, um seine Ausgangsereignisse zu empfangen. Die `@Map`-Annotation mit dem definierten Typ `passThrough` sorgt dafür, dass die Ereignisse ohne jegliche Transformation des Datenformats weitergeschickt werden, d. h. Ereignisse besitzen das standardmäßige Ereignisformat von Siddhi (siehe Abschnitt 3.4).

```
1 @source(type='inMemory', topic='CleanedPrEvents',@map(type='passThrough'))
2 define stream CleanedPulseRateEventStream (eventType String, timestamp
      String, deviceId int, patientId int, age int, pulseRate int, deviceUnit
3      String, eventSource String);
```

Auflistung 4.2: Empfängt Ereignisse von anderen EPA-Instanzen in Siddhi

Der `CleanedPulseRateEventStream` der anderen EPA-Instanz, die das Thema abonniert, besitzt das identische Ereignisschema zum Publisher (Zeile 2-3) und eine `@Source`-Annotation wie Code-Beispiel 4.2 (Zeile 1). Die `@Source`-Annotation enthält die gleichen Parameter wie die der `@Sink`-Annotation des Publishers und empfängt die Ausgangsereignisse. Diese Ausgangsereignisse können daraufhin von der weiteren Ereignisanfrage innerhalb der EPA-Instanz verarbeitet werden.

Die Ereignisregel in Code-Beispiel 4.1 dient an diesem Punkt nur zum Demonstrationszweck und wird in Abschnitt 4.3.2 genauer erklärt.

Das Code-Beispiel 4.1 wurde für die beiden Ereignisströme `PulseRateEventStream` und `CleanedPulseRateEventStream` vorgestellt. Da die Umsetzung des `BloodPressureEventStream` und des `CleanedBloodPressureEventStream` analog verläuft, wurde dies aus Gründen der Redundanz ausgelassen.

```
1  SiddhiManager sm = new SiddhiManager();
2  SiddhiAppRuntime sar = sm.createSiddhiAppRuntime(inputStream+outputStream+
       query);
3
4  sar.addCallback("CleanedPulseRateEventStream", new StreamCallback(){
5      @Override
6      public void receive(Event[] events) {
7          EventPrinter.print(events);
8          //hier weitere Aufrufe von Methoden fuer Ereignisbehandlung
               moeglich
9      }
10 });
```

Auflistung 4.3: Erstellung der Anwendungslaufzeit und des Stream Callbacks in Siddhi

In Abschnitt 3.2 wurde die Funktion des `Siddhi Managers` und die `Siddhi Application Runtime` bereits erläutert. In Code-Beispiel 4.3 wird mithilfe der `createSiddhiAppRuntime`-Methode des `SiddhiManagers` das Laufzeitobjekt vom Typ `SiddhiAppRuntime` erstellt und der entsprechende String aus Code-Beispiel 4.1, der die definierten Ereignisströme und Ereignisregeln beinhaltet, übergeben (Zeile 2). Der String wird daraufhin intern vom Siddhi Query Compiler verarbeitet. Der Siddhi Query Compiler ist, wie ebenfalls in Code-Beispiel 4.1 besprochen, für die Konvertierung der definierten Ereignisströme und Ereignisregeln in Prozessoren zuständig. Das erstellte `SiddhiAppRuntime`-Objekt kann anschließend zum Hinzufügen eines Callbacks genutzt werden. In dieser Umsetzung wird dazu der `Stream Callback` genutzt. Der `Stream Callback` wird mit der `addCallback`-Methode bei der Siddhi Anwendungslaufzeit registriert (Zeile 4), um verarbeitete `PulseRateEvents` des abonnierten `CleanedPulseRateEventStreams` zu erhalten. Innerhalb dieser Methode wird die Methode `receive` aufgerufen (Zeile 6), sobald ein `PulseRateEvent` im `CleanedPulseRateEventStream` eintrifft. Anschließend werden die `PulseRateEvents` mit der `print`-Methode des `EventPrinter`-Objektes ausgegeben (Zeile 7). Die `receive`-Methode kann ebenso zum Anstoßen von weiteren Aktionen genutzt werden wie z. B. der Aufruf einer weiteren Methode zur Ereignisbehandlung.

4.3.2. Umsetzung der Ereignisregeln

In Abschnitt 4.2 wurde erklärt, welche Fachlichkeit die Ereignisregeln realisieren sollen. Aus diesem Grund wird im Folgenden nur auf die Sprachelemente der Ereignisregeln mit der Siddhi Streaming SQL eingegangen.

1. Ereignisregel

```
1 from PulseRateEventStream[pulseRate > 40 and pulseRate < 130]
2 select *
3 insert into CleanedPulseRateEventStream;
4
5 from BloodPressureEventStream[systolicPressure > 80 and
6 systolicPressure < 280 and diastolicPressure > 40 and
7 diastolicPressure < 200]
8 select *
9 insert into CleanedBloodPressureEventStream;
```

Auflistung 4.4: Filtert Ereignisse aus dem `PulseRateEventStream` und `BloodPressureEventStream` nach den festgelegten Grenzen im Bedingungsteil und erzeugt neue Ereignisse für den `CleanedPulseRateEventStream` und `CleanedBloodPressureEventStream`.

In Siddhi werden Filteranfragen genutzt, um Informationen nach angegebener Bedingung aus einem Ereignisstrom zu filtern. Ereignisse werden dadurch von anderen Ereignissen getrennt und können zur Ausgabe oder weiteren Verarbeitung weitergeleitet werden. Die Bedingung wird wie in Code-Beispiel 4.4 zu erkennen, neben der Definition des Ereignisstroms in eckigen Klammern festgelegt. Die Bedingungen können in Siddhi aus den in Abschnitt 2.3.3 beschriebenen Vergleichs-, Konjunktions- oder Disjunktionsoperatoren zusammengesetzt werden. In diesem Fall wurden die beiden Vergleichsoperatoren *Kleiner-als-Operator* `<` und *Größer-als-Operator* `>` genutzt, um die Grenzen der erlaubten Attributwerte von `pulseRate` bzw. `systolicPressure` und `diastolicPressure` zu setzen (Zeile 1 und 5-7). Mithilfe der `SELECT`-Klausel der Ereignisanfrage lassen sich nur die Attribute eines eintreffenden Ereignisses auswählen, die von weiterer Relevanz sind, und werden anschließend als Ereignis in den Ausgangsstrom hinzugefügt. In der `SELECT`-Klausel der Ereignisregel wird mit dem `*`-Platzhalter (auch Wildcard genannt) ausgedrückt (Zeile 2 und 8), dass der zugrundeliegende Ereignistyp des Ereignisstroms weitergeleitet wird, d. h. das komplette `PulseRateEvent` mit seinen spezifischen Ereignisattributen. Anschließend werden alle gefilterten `PulseRateEvents` bzw. `BloodPressureEvents` mit `INSERT INTO` in ihren entsprechenden Ausgangsstrom `PulseRateEventStream` bzw. `BloodPressureEventStream` hinzugefügt (Zeile 3 und 9).

2. Ereignisregel

```
1 from CleanedPulseRateEventStream#window.time(1 sec) as p join
2      CleanedBloodPressureEventStream#window.time(1 sec) as b
3      on p.timestamp == b.timestamp
4 select p.patientId as patientId, p.age as age, avg(p.pulseRate) as avgPr,
5        p.deviceUnit as deviceUnitPr, avg(b.systolicPressure) as avgBpSys,
6        avg(b.diastolicPressure) as avgBpDia, b.deviceUnit as deviceUnitBp
7 group by p.patientId
8 insert into AverageHealthDataEventStream;
```

Auflistung 4.5: Kombiniert Ereignisse aus `CleanedPulseRateEventStream` und `CleanedBloodPressureStream` und berechnet die Durchschnittswerte der einzelnen Vitalwerte während eines Zeitfensters.

Mithilfe des Schlüsselworts `JOIN` können in Siddhi die beiden Ereignisströme `CleanedPulseRateEventStream` und `CleanedBloodPressureEventStream` vereint werden, um mehrere verknüpfte Ereignisse zu erhalten. Das Schlüsselwort `JOIN` wird dabei in der `FROM`-Klausel zwischen den zu kombinierenden Ereignisströmen gesetzt (Zeile 1). Da die beiden Ereignisströme gleiche Attribute enthalten, müssen diese mit dem Schlüsselwort `AS` umbenannt werden, um später in der `SELECT`-Klausel zu unterscheiden, welches Attribut von welchem Ereignisstrom selektiert wird (Zeile 1-2). Meistens empfiehlt es sich, kurze Bezeichner zu definieren, um die Ereignisregeln übersichtlich zu halten.

Im Allgemeinen sind Ereignisströme zustandslos, daher müssen die beiden Ereignisströme ein Sliding Time Window (auch andere Fensterart möglich) zur Verknüpfung verwenden. In diesem Fall speist das festgelegte Zeitfenster alle `PulseRateEvents` und `BloodPressureEvents` ein, die in der ersten Sekunde eintreffen. Ein Sliding Time Window wird in Siddhi nach der Definition des Ereignisstroms in der `FROM`-Klausel mit `#window` gesetzt (Zeile 1-2). Um das Zeitfenster wie in Code-Beispiel 4.5 zu nutzen, wird nach einer gefolgten Punktnotation, die `time()`-Funktion des Windows aufgerufen. Innerhalb der Zeitfensterklammern kann daraufhin die Zeit definiert werden. In dieser Ereignisregel wurde der Zeitraum auf eine Sekunde gesetzt, um alle eingehenden Ereignisse zu konsumieren. Durch das Sliding Time Window entsteht nun ein Ereignispool der beiden Ereignisströme. Der `JOIN` wird hier als der bekannte *Inner Join* aus der Datenbanksprache SQL interpretiert, d. h. mindestens ein Ausgabeereignis für jedes Ereignis aus dem `CleanedPulseRateEventStream` auf der linken Seite, muss mit mindestens einem Ereignis aus dem `CleanedBloodPressureEventStream` auf der rechten Seite übereinstimmen. Dabei wird die Verbindungsbedingung berücksichtigt, die mit dem `ON`-Schlüsselwort angegeben wird (Zeile 3). Um die Verbindungsbedingung zu erfüllen, muss der `timestamp` eines Ereignisses aus dem `CleanedPulseRateEventStream` mit dem `timestamp` eines Ereignisses aus dem `CleanedBloodPressureEventStream` übereinstimmen. Auf jedem übereinstimmenden Ereignispaar von `PulseRateEvents` und `BloodPressureEvents` wird anschließend eine Aggregation durchgeführt, um den Durchschnittswert der Blutdruckwerte und der Pulswerte jedes Patienten zu berechnen. Siddhi

stellt dafür die vorgefertigte `avg()`-Funktion bereit. Die `avg()`-Funktion bekommt die entsprechenden Attribute der Ereignisse übergeben (Zeile 4-6): für die `PulseRateEvents` das Attribut `pulseRate` und für die `BloodPressureEvents` jeweils `systolicPressure` und `diastolicPressure`. Damit die berechneten Attribute mit den Attributsnamen des Ausgangsstroms `AverageHealthDataEventStream` übereinstimmen, können diese ebenfalls mit `AS` umbenannt werden. Mithilfe der `GROUPBY`-Klausel kann dann die Aggregation basierend auf das angegebene Attribut `patientId` gruppiert werden (Zeile 7).

3. Ereignisregel

```
1 from every (e1 = AverageHealthDataEventStream) ->
2             e2 = AverageHealthDataEventStream[e1.patientId == patientId
3             and (e1.avgPr + 25) <= avgPr]
4 select e1.patientId as patientId, e1.age as age, e1.avgPr as avgPr1,
5        e2.avgPr as avgPr2, e2.deviceUnitPr as deviceUnit
6 insert into AbnormalityEventStream;
```

Auflistung 4.6: Erkennt, wenn ein Ereignis aus dem `PulseRateEventStream` eintrifft, dessen durchnittlicher Pulswert (`avgPr`) eines Patienten um 25 oder mehr Pulsschläge pro Minute im Vergleich zum vorhergehenden Ereignis höher ist.

Wie in Code-Beispiel 4.6 zu erkennen, lassen sich in Siddhi Ereignismuster mit dem Followed-By-Operator (`->`), auch Sequenzoperator (siehe Abschnitt 2.3.3) genannt, realisieren (Zeile 2). Dabei ist zu beachten, dass die Bedingung, die das vorhergehende Ereignis erfüllen muss, vor dem Followed-By-Operator hinzugefügt wird und die Bedingung, die das nachfolgende Ereignis erfüllen muss, nach dem Followed-By-Operator hinzugefügt wird. In diesem Fall wird der Ereignisstrom `AverageHealthDataEventStream` vor dem Followed-By-Operator definiert und bekommt zusätzlich das Schlüsselwort `every` davor gesetzt. Das Schlüsselwort `every` sorgt dafür, dass der Abgleich des Musters für jedes eintreffende `AverageHealthDataEvent` vorgenommen wird. Dennoch ist `every` optional, sodass der Abgleich nur einmal ausgeführt wird, wenn es nicht gesetzt ist. Nach dem Followed-By-Operator wird nun ebenfalls der `AverageHealthDataEventStream` definiert. In den eckigen Klammern wird die Musterbedingung für das nachfolgende `AverageHealthDataEvent` festgelegt (Zeile 2-3). Da in dieser Ereignisregel ebenfalls auf die Attribute des jeweiligen `AverageHealthDataEventStream` zugegriffen werden muss, werden bei einer Musteranfrage die beiden `AverageHealthDataEventStream` in unterschiedlichen Variablen initialisiert. In der Bedingung sowie in der `SELECT`-Klausel kann nun unterschieden werden, welches Attribut von welchem `AverageHealthDataEventStream` gemeint ist.

4. Ereignisregel

```
1 from PulseRateEvent[age > 18 and age < 80]
2 select *
3 insert into outputStream;
```

Auflistung 4.7: Filtert die eintreffenden Ereignisse aus dem `PulseRateEventStream`, deren Alter der Patienten zwischen 18 und 80 Jahren liegt.

Die Sprachelemente dieser Filteranfrage in Code-Beispiel 4.7 sind identisch mit der Ereignisregel in Code-Beispiel 4.4. Die Bedingung der Ereignisregel legt die Grenze der Attributwerte für `age` zwischen 18 und 80 fest (Zeile 1). Anschließend werden nur die `PulseRateEvents` ausgegeben, die der Bedingung entsprechen.

5. Ereignisregel

```
1 from PulseRateEventStream#window.lengthBatch(5)
2 select patientId, max(age) as age
3 insert into outputStream;
```

Auflistung 4.8: Ermittelt aus eintreffenden Ereignissen des `PulseRateEventStreams` den Patienten mit dem höchsten Alter innerhalb eines Batch Length Window.

Die fünfte Ereignisregel ist vergleichbar mit der zweiten Ereignisregel aus Code-Beispiel 4.5 und realisiert ebenfalls eine Aggregationsanfrage, verwendet aber stattdessen eine `max`-Funktion (Zeile 2). Die `max`-Funktion wird ebenso von Siddhi bereitgestellt und bekommt das Attribut `age` übergeben, um das höchste Alter eines Patienten aus einem Batch Length Window zu ermitteln. Die Länge des Windows wird dabei in `lengthBatch` übergeben, sodass die `max`-Funktion immer auf 5 eintreffende `PulseRateEvents` ausgeführt wird. Das resultierende Ereignis enthält das Attribut `patientId` als auch das Attribut `age` und wird anschließend in den `outputStream` hinzugefügt und ausgegeben.

6. Ereignisregel

```
1 from every e1=PulseRateEventStream ->
2           e2=PulseRateEventStream[e1.pulseRate < pulseRate]
3 select e1.patientId, e1.pulseRate, e2.pulseRate
4 insert into outputStream;
```

Auflistung 4.9: Erkennt, wenn ein Ereignis aus dem `PulseRateEventStream` eine höhere `pulseRate` besitzt als das vorhergehende Ereignis.

Die letzte Ereignisregel benutzt auch den Followed-By-Operator (`->`), um ein Muster festzulegen, wie es bereits für die dritte Ereignisregel in Abschnitt 4.3.2 erklärt wurde. Wie in Code-Beispiel 4.9 wird das Schlüsselwort `every` benutzt, sodass das Muster für jedes eintreffende `PulseRateEvent` abgeglichen wird (Zeile 1). Die Bedingung für das nachfolgende `PulseRateEvent` aus dem `PulseRateEventStream` wird wieder in den eckigen Klammern festgelegt und drückt aus, dass die `pulseRate` größer als die `pulseRate` des vorherigen `PulseRateEvent` sein muss.

4.4. Realisierung mit Esper

In diesem Abschnitt wird die Umsetzung der Fallstudie mit der CEP Engine Esper vorgestellt. Da Esper in dieser Arbeit als Vergleichskandidat herangezogen wurde, soll bei der Umsetzung nur auf das Wesentliche eingegangen werden. Weitere relevante Informationen können daher in der Esper Dokumentation [Esp18f] nachgelesen werden.

4.4.1. Implementation der Esper-Anwendung

In Esper wurden die Ereignistypen mithilfe von POJO mit Getter-/Setter-Methoden, die den JavaBean-Konventionen folgen, implementiert. Zusätzlich gibt es auch weitere Möglichkeiten wie z. B. per XML-Dokument. Im Folgenden soll auf die Erzeugung eines EPA in der Esper-Anwendung eingegangen werden und wie sich die definierte Ereignisregeln in einem EPA registrieren lassen.

```
1  ClassLoader classLoader = FilterEventAgent.class.getClassLoader();
2  String rootPath = classLoader.getResource("files").getPath();
3
4  Configuration config = new Configuration();
5  config.addEventType(PulseRateEvent.class);
6  EPServiceProvider epa =
7                      EPServiceProviderManager.getProvider("epa", config);
8
9  EPStatement eventRule1 =
10                     epa.getEPAdministrator().createEPL(
11                     "select * from PulseRateEvent where pulseRate > 40
12                     and pulseRate < 130");
13
14 eventRule1.setSubscriber(new PulseRateEventSubscriber());
15 (new CSVInputAdapter(epa, new AdapterInputSource(new URL("FILE://" +
       rootPath + "/pulse_testData.csv")), "PulseRateEvent")).start();
```

Auflistung 4.10: Erstellung eines EPA und einer Ereignisregel in Esper

Das Code-Beispiel 4.10 zeigt die Erzeugung und Initialisierung eines Esper-EPA. Zuerst muss der Esper-Anwendung die Java-Ereignisklasse mithilfe der `Configuration`-Klasse bekannt gemacht werden. Dafür wird die `addEventType`-Methode des `Configuration`-Objekts benutzt. In diesem Fall wird der `addEventType`-Methode die Ereignisklasse `PulseRateEvent.class` übergeben, um den Ereignistyp beim EPA zu registrieren (Zeile 5). Die Engine prüft anhand der Getter-/Setter-Methoden der `PulseRateEvent`-Klasse, welche Ereignisattribute der Ereignistyp `PulseRateEvent` besitzt. Danach wird der `getProvider`-Methode des `EPServiceProviderManagers` der Name des EPA als String und die vorher erstellte Konfiguration `config` übergeben, um den EPA unter der Managerklasse bekannt zu machen (Zeile 6-7). Die erzeugte EPA-Instanz ist nun zur weiteren Verwendung unter dem Namen `epa` referenzierbar. Um im Weiteren der EPA-Instanz `epa` eine Ereignisregel zuzuweisen, muss die Schnittstelle des `EPServiceProvider` verwendet werden. Dafür ruft `epa` die Methode `getEPAdministrator` auf (Zeile 10). Die `EPAdministrator`-Klasse stellt die `createEPL`-Methode bereit, welche die Ereignisregel als String übergeben bekommt und diese daraufhin bei der EPA-Instanz `epa` registriert. Die Ereignisse werden in Code-Beispiel 4.10 mithilfe der `CSVInputAdapters`-Klasse aus einer CSV-Datei gelesen. Damit dies erfolgt, muss bei Erzeugung des `CSVInputAdapter`-Objekts, die EPA-Instanz sowie ein `AdapterInputSource`-Objekt, welches die URL der Datei und den Ereignistyp `PulseRateEvent` als Parameter enthält, übergeben und anschließend gestartet werden (Zeile 15).

Bei Erfüllung der Ereignisregel lassen sich Ereignisse ausgegeben oder sie stoßen eine weitere Methode an. Dazu muss die Ereignisregel sich an einem sogenannten *Subscriber* registrieren. In Esper werden Ereignisregeln als `EPStatement`-Objekte erzeugt (Zeile 9), welche die Methode `setSubscriber` bereit stellen. Die `setSubscriber`-Methode bekommt anschließend das Subscriber-Objekt `new PulseRateEventSubscriber()` übergeben (Zeile 14).

```
1 public class PulseRateEventSubscriber {
2     public void update(PulseRateEvent event) {
3         System.out.println(event.getPulseRate());
4         //hier Auruf von weiteren Methoden zur Ereignisbehandlung moeglich
5     }
6 }
```

Auflistung 4.11: Erstellung der Subscriber-Klasse in Esper

Die Subscriber-Klasse `PulseRateEventSubscriber` stellt eine `update`-Methode bereit. Mit der `update`-Methode können genau wie die `receive`-Methode des Siddhi Callbacks aus Code-Beispiel 4.3, jegliche Aktionen angestoßen werden. In diesem Fall gibt die `update`-Methode demonstrativ die Pulswerte der eintreffenden `PulseRateEvents` aus, die das Muster der Ereignisregel erfüllen. Die übergebenen Parameter der `update`-Methode müssen mit ausgewählten Ereignisattributen der `SELECT`-Klausel und der registrierten Ereignisregel übereinstimmen. Da die `SELECT`-Klausel der Ereignisregel in

Code-Beispiel 4.10 einen *-Platzhalter enthält und damit den zugrundeliegenden Ereignistyp des Ereignisstroms meint, wird das komplette Ereignis mit den entsprechenden Attributen ausgegeben. Aus diesem Grund kann als Parameter der `update`-Methode, der Ereignistyp bzw. das Java-Objekt `PulseRateEvent` übergeben werden.

Die Ereignisregel in Code-Beispiel 4.10 dient ebenfalls nur zur Veranschaulichung und wird im folgenden Code-Beispiel 4.12 genauer betrachtet.

4.4.2. Umsetzung der Ereignisregeln

Die meisten Sprachelemente, Operatoren und Aggregationsfunktionen von Esper sind identisch zu denen in Siddhi, daher wird im Folgenden nur auf mögliche Unterschiede eingegangen.

1. Ereignisregel

```
1  insert into CleanedPulseRateEvent
2  select *
3  from PulseRateEvent
4  where pulseRate > 40 and pulseRate < 130
5
6  insert into CleanedBloodPressureEvent
7  select *
8  from BloodPressureEvent
9  where systolicPressure > 80 and systolicPressure < 280 and
10       diastolicPressure > 40 and diastolicPressure < 200
```

Auflistung 4.12: Filtert `PulseRateEvents` und `BloodPressureEvents` nach den festgelegten Grenzen im Bedingungsteil und erzeugt neue `CleanedPulseRateEvents` und `CleanedBloodPressureEvents`.

In Esper wird jeweils die Bedingung der beiden Filterereignisregeln aus Code-Beispiel 4.12 in der `WHERE`-Klausel festgelegt (Zeile 4 und 9). Diese Regel filtert ebenfalls die `pulseRates` und `systolicPressures` bzw. `diastolicPressures` nach den definierten Grenzen. Durch `INSERT INTO` werden anschließend aus den gefilterten `PulseRateEvents` und `BloodPressureEvents` neue `CleanedPulseRateEvents` und `CleanedBloodPressureEvents` erzeugt (Zeile 1 und 6).

2. Ereignisregel

```
1 insert into AverageHealthDataEvent(patientId, age, avgPr, deviceUnitPr,
2     avgBpSys, avgBpDia, deviceUnitBp)
3 select p.patientId as patientId, p.age as age, avg(p.pulseRate) as avgPr,
4        p.deviceUnit as deviceUnitPr,avg(b.systolicPressure) as avgBpSys,
5        avg(b.diastolicPressure) as avgBpDia, b.deviceUnit as deviceUnitBp
6 from CleanedPulseRateEvent#time(1 sec) as p join
7      CleanedBloodPressureEvent#time(1 sec) as b
8      on p.timestamp = b.timestamp
9 group by p.patientId
```

Auflistung 4.13: Kombiniert `CleanedPulseRateEvents` und `CleanedBloodPressureEvents` aus ihren jeweiligen Ereignisströmen und berechnet die Durchschnittswerte der einzelnen Vitalwerte während eines Sliding Time Windows.

Die zweite Ereignisregel in Esper aus Code-Beispiel 4.13 führt die Aggregationsfunktion auf den beiden verknüpften `CleanedPulseRateEvents` und `CleanedBloodPressureEvents` aus. Esper unterstützt ein Inner Join auf die gleiche Art wie Siddhi mithilfe des Schlüsselwortes `JOIN`, das zwischen den beiden zu verknüpfenden Ereignistypen in der Regel definiert wird (Zeile 6). Esper erzeugt bei einem Inner Join analog zu Siddhi ein Ausgabeereignis für jedes `CleanedPulseRateEvent` des Ereignisstroms auf der linken Seite, das mit mindestens einem `CleanedBloodPressureEvent` auf der rechten Seite übereinstimmt. Dabei wird die Verbindungsbedingung (`timestamp` des `CleanedPulseRateEvent` und des `CleanedBloodPressureEvent` stimmen überein) berücksichtigt, die mit dem `ON`-Schlüsselwort angegeben wird (Zeile 8). Um festzulegen, dass nur `CleanedPulseRateEvents` und `CleanedBloodPressureEvents` eingespeist werden, die innerhalb eines Sliding Time Windows von einer Sekunde eintreffen, wird dieses mit `#time(1 sec)` hinter beiden Ereignistypen definiert (Zeile 6-7). Die Aggregationsberechnung wird anschließend mit der `avg()`-Funktion (Zeile 3-5), welche ebenfalls von Esper bereitgestellt wird, für jeden Patienten (`GROUPBY`-Klausel) gruppiert durchgeführt. Dazu werden die entsprechenden Durchschnittsvitalwerte `avgPr`, `avgBpSys` und `avgBpDia` der `avg()`-Funktion übergeben. Die resultierenden Ausgangsereignisse werden in den `AverageHealthDataEvent`-Ereignisstrom übernommen.

3. Ereignisregel

```
1 insert into AbnormalityEvent(patientId, age, avgPr1, avgPr2, deviceUnit)
2 select e1.patientId as patientId, e1.age as age, e1.avgPr as avgPr1,
3        e2.avgPr as avgPr2, e1.deviceUnitPr as deviceUnit
4 from pattern [every e1=AverageHealthDataEvent ->
5                     e2=AverageHealthDataEvent(e1.patientId=patientId
6                     and (e1.avgPr + 25) <= avgPr)]
```

Auflistung 4.14: Erkennt, wenn ein `PulseRateEvent` eintrifft, dessen durchnittlicher Pulswert (`avgPr`) eines Patienten um 25 oder mehr Pulsschläge pro Minute im Vergleich zum vorhergehenden `PulseRateEvent` angestiegen ist.

Die dritte Esper Ereignisregel erzeugt `AbnormalityEvents` aus den eintreffenden `AverageHealthDataEvents`, die das Ereignismuster erfüllen. Um das Ereignismuster festzulegen, wird in Esper das Schlüsselwort `pattern` verwendet (Zeile 4). Alle weiteren Operatoren und Schlüsselwörter haben dabei die gleiche Funktionalität wie die in der dritten Siddhi-Ereignisregel aus Code-Beispiel 4.6.

Die nachfolgenden Ereignisregeln in Esper werden im Einzelnen nicht weiter erklärt, da bereits alle enthaltenen Sprachelemente und Operatoren in vorherigen Esper- und Siddhi-Ereignisregel erklärt wurden und keinen neuen Inhalt leisten.

4. Ereignisregel

```
1 select *
2 from PulseRateEvent
3 where age > 18 and age < 80
```

Auflistung 4.15: Filtert die `PulseRateEvents` von Patienten zwischen 18 und 80 Jahren.

5. Ereignisregel

```
1 select patientId, max(age) as maxAge
2 from PulseRateEvent#length_batch(5)
```

Auflistung 4.16: Ermittelt aus eintreffenden `PulseRateEvents` den Patienten mit dem höchsten Alter innerhalb eines Batch-Längenfensters.

6. Ereignisregel

```
select e1.patientId, e1.pulseRate, e2.pulseRate
from pattern [every (e1=PulseRateEvent ->
    e2=PulseRateEvent(e1.pulseRate < pulseRate))]
```

Auflistung 4.17: Erkennt, wenn ein `PulseRateEvent` eine höhere `pulseRate` besitzt, als das vorhergehende `PulseRateEvent`.

5. Vergleich und Evaluierung der ereignisverarbeitenden Engines Siddhi und Esper

In diesem Kapitel werden die beiden CEP Engines Siddhi und Esper mithilfe von sprachlichen, konzeptionellen und technischen Kriterien untersucht und verglichen. Der Kriterienkatalog wurde anhand von ausgewählten Kriterien aus einer Evaluierungscheckliste für CEP Engines vom Softwareunternehmen TIBCO [Vin10] und aus den Qualitätsanforderungen an eine Software nach ISO/IEC 25010:2011 [Int11] aufgestellt. Am Ende des Kapitels werden Siddhi und Esper jeweils nach den aufgestellten Kriterien in der Tabelle 5.1 evaluiert.

5.1. Sprachlicher und konzeptioneller Vergleich

5.1.1. Kernoperatoren & Aggregationsfunktionen

Das Kriterium *Kernoperatoren & Aggregationsfunktionen* soll die grundlegenden Operatoren und Aggregationsfunktionen der beiden Engines aufstellen, die bei der Umsetzung von Ereignisregeln verwendet werden können.

Siddhi

Bei der Umsetzung der Fallstudie mit Siddhi in Abschnitt 4.3.2 wurden einige Kernoperatoren und Aggregationsfunktionen zur Definition der Ereignisregeln verwendet und deren Auswirkung erklärt. In Siddhi stehen ebenso jegliche Kernoperatoren, die in Abschnitt 2.3.3 beschrieben wurden, zur Verfügung. Im Folgenden sollen noch weitere grundlegende Kernoperatoren sowie Aggregationsfunktionen aufgelistet werden, die in Siddhi verfügbar sind und in dieser Arbeit noch nicht erwähnt wurden.

Vergleichsoperatoren und mathematische/logische Operatoren:

- **Mathematische Operatoren** (+, -, %, *, /) für Additions-, Subtraktions-, Modulo-, Multiplikations- und Divisionsberechnungen.

- Der **IS NULL-Operator** zur Überprüfung, ob ein Ereignisattribut einen fehlenden unbekannten Wert (NULL) besitzt. [WSO18l]

Aggregationsfunktionen:

- Mit **count()** kann die Anzahl der Ereignisse ausgegeben werden.
- Mit **distinctCount(<Attribut>)** kann die Anzahl der verschiedenen Vorkommen eines spezifischen Ereignisattributs ausgeben werden. [WSO18l] So würde beispielsweise `distinctCount(temp)` die Anzahl 3 zurückgeben, wenn Temperaturereignisse mit den Werten `25.6; 18.2; 25.6; 20.1` eintreffen würden.
- Mit **maxForever(<Attribut>)/minForever(<Attribut>)** kann der Maximalwert bzw. Mindestwert des übergebenen Ereignisattributs während der kompletten Ereignisverarbeitung und unabhängig vom festgelegten Window gespeichert werden. [WSO18l]
- Mit **stdDev(<Attribut>)** kann die Standardabweichung des übergebenen Attributs für alle eintreffenden Ereignisse berechnet werden. [WSO18l]

Neben den Kernoperatoren und Aggregationsfunktionen existieren noch weitere hilfreiche Funktionen, z. B. zur Überprüfung von Attributstypen oder zur Konvertierung von Attributen in einen anderen Attributstyp (Integer, Double, String, ...), die aber an dieser Stelle nicht weiter erwähnt werden, da die Aufzählung und Beschreibung der Funktionen den Inhalt der Arbeit sprengen würden. [WSO18l] Bei weiterem Interesse kann auf Siddhis Dokumentation[1] zurückgegriffen werden.

Esper

In Abschnitt 4.4.2 wurden ebenso einige Kernoperatoren und Funktionen in Espers Ereignisregeln eingesetzt. Auch Esper stellt alle erwähnten Kernoperatoren aus Abschnitt 2.3.3 bereit sowie die bereits vorgestellten Siddhi Kernoperatoren und Aggregationsfunktionen. Im Folgenden soll daher auf weitere grundlegende Kernoperatoren von Esper eingegangen werden, die bisher nicht erwähnt wurden oder unterschiedlich zu Siddhi sind.

Vergleichsoperatoren und logische Operatoren:

- Der **||-Operator** zur Verkettung von Zeichenfolgen. [Esp18f]
- Die **Binary**-Operatoren, z. B. **&** für bitweises AND, **|** für bitweises ODER und **^** für exklusives ODER (XOR). [Esp18f]
- Der **IS/IS NOT-Operator** zur Überprüfung, ob ein Ereignisattribut gleich bzw. nicht gleich einem Wert/Ereignisattribut ist. [Esp18f]

[1] Siddhis Dokumentation unter https://wso2.github.io/siddhi/api/latest/

Aggregationsfunktionen:

- Mit **avedev(<Attribut>)** kann die mittlere Abweichung des übergebenen Attributs für alle eintreffenden Ereignisse berechnet werden. [Esp18f]
- Mit **median(<Attribut>)** kann der Mittelwert des übergebenen Attributs für alle eintreffende Ereignisse berechnet werden. [Esp18f]

Esper hat den Vorteil, dass Filterausdrücke für alle verfügbaren Esper-Funktionen direkt als Parameter in der Funktion übergeben werden können [Esp18f], z. B. würde `select avg(temp, roomNu=2) from TempEvent` nur die Durchschnittstemperatur für eintreffenden Ereignisse mit der Raumnummer zwei berechnen.

Im Vergleich zu Siddhi besitzt Esper ein riesiges Spektrum an nützlichen Funktionen, die ebenfalls nicht komplett aufgeführt werden. An dieser Stelle lässt sich hier auf Espers Dokumentation[2] verweisen.

5.1.2. Windows

Anhand des Kriteriums *Windows* sollen die grundlegenden Fensterarten aus Abschnitt 2.3.4 untersucht werden, welche Siddhi und Esper bereitstellen.

Siddhi

In Abschnitt 4.3 wurden bereits Windows in Siddhis Ereignisregeln genutzt, sodass jeder Ereignisstrom über ein eigenes Window verfügte. In Siddhi gibt es eine weitere Variante, um verschiedene Windows zu erstellen. Diese Windows werden auch *Named Windows* genannt und lassen sich von mehreren Ereignisanfragen benutzen. Die Named Windows werden ähnlich wie Ereignisströme (mit `define`) erstellt und können auch unterschiedliche Fensterarten realisieren. [WSO18a] Im Folgenden wird auf die Erstellung von Named Windows mit den grundlegenden Fensterarten (Zeitfenster und Längenfenster) eingegangen. Durchaus bietet Siddhi noch weitere Arten von Named Windows an.

[2] Espers Dokumentation unter http://esper.espertech.com/release-7.1.0/esper-reference/html/index.html

Sliding Time Window

```
1 define window TempEventWindow (sensorID string, roomNu int, temp double)
    time(40 sec) output current events;
2
3 from TempEventStream
4 select *
5 insert into TempEventWindow;
6
7 from TempEventWindow
8 select avg(temp) as avgTemp
9 insert into outputStream;
```

Auflistung 5.1: Ereignisanfrage mit einem Sliding Time Window als Named Window in Siddhi

Das Code-Beispiel 5.1 stellt ein Sliding Time Window dar und wird mit der `time()`-Funktion realisiert (Zeile 1). [WSO18a] Es enthält alle `TempEvents`, die innerhalb der letzten 40 Sekunden eintreffen. Anschließend wird mit der zweiten Ereignisanfrage (Zeile 7-9) für alle aktuellen `TempEvents` im Window die Durchschnittstemperatur ausgerechnet und dem `outputStream` hinzugefügt. Am Ende der Definition des Windows kann angegeben werden, ob aktuelle Ereignisse (`output current events`), abgelaufene Ereignisse (`output expired events`) oder alle Ereignisse (`output all events`) ausgegeben werden sollen. Bei keiner Angabe, wird standardmäßig `output all events` verwendet.

Batch Time Window

```
1 define window TempEventWindow (sensorID string, roomNu int, temp double)
    timeBatch(40 sec) output current events;
2
3 from TempEventStream
4 select *
5 insert into TempEventWindow;
6
7 from TempEventWindow
8 select avg(temp) as avgTemp
9 insert into outputStream;
```

Auflistung 5.2: Ereignisanfrage mit einem Batch Time Window als Named Window in Siddhi

Das Code-Beispiel 5.2 zeigt ein Batch (Tumbling) Time Window, welches mit der `timeBatch()`-Funktion umgesetzt wird (Zeile 1). [WSO18a] Es verarbeitet alle `TempEvents` als Batch (Stapel), die jede 40 Sekunden eintreffen. Die zweite Ereignisanfrage (Zeile 7-9) berechnet ebenfalls die Durchschnittstemperatur der aktuellen `TempEvents` und leitet

diese an den `outputStream` weiter. Zusätzlich kann eine Versatzzeit in Millisekunden an `timeBatch()` übergeben werden, falls das Window zu einer bestimmten Zeit starten soll. Üblicherweise nutzt es als Startzeit den Zeitstempel des ersten eintreffenden Ereignisses.

Sliding Length Window

```
1 define window TempEventWindow (sensorID string, roomNu int, temp double)
    length(20) output current events;
2
3 from TempEventStream
4 select *
5 insert into TempEventWindow;
6
7 from TempEventWindow
8 select avg(temp) as avgTemp
9 insert into outputStream;
```

Auflistung 5.3: Ereignisanfrage mit einem Sliding Length Window als Named Window in Siddhi

Das Code-Beispiel 5.3 repräsentiert ein Sliding Length Window, welches sich mit der `length()`-Funktion umsetzen lässt (Zeile 1). [WSO18a] Im Gegensatz zu den Sliding Time Windows enthält es eine übergebene Anzahl an Ereignissen. Im Code-Beispiel 5.3 werden daher immer die letzten 20 `TempEvents` eingespeist, d. h. trifft ein neues `TempEvent` ein, wird das älteste `TempEvents` aus dem Window entfernt (FIFO-Prinzip). Auf diesen letzten 20 `TempEvents` wird mit der zweiten Ereignisanfrage (Zeile 7-9) wieder die durchschnittliche Temperatur ausgerechnet und anschließend dem `outputStream` hinzugefügt.

Batch Length Window

```
1 define window TempEventWindow (sensorID string, roomNu int, temp double)
    lengthBatch(20) output current events;
2
3 from TempEventStream
4 select *
5 insert into TempEventWindow;
6
7 from TempEventWindow
8 select avg(temp) as avgTemp
9 insert into outputStream;
```

Auflistung 5.4: Ereignisanfrage mit einem Batch Length Window als Named Window in Siddhi

Das letzte Code-Beispiel 5.4 stellt ein Batch (Tumbling) Length Window dar und verwendet dafür die `lengthBatch()`-Funktion (Zeile 1). [WSO18a] Sobald das Window die exakte Anzahl an `TempEvents` enthält, die in `lengthBatch()` übergeben wurden, wird es komplett geleert und sammelt in der nächsten Fensterinstanz die weiteren 20 eintreffenden `TempEvents`. Für jeden Batch an `TempEvents` wird ebenfalls durch die zweite Ereignisanfrage (Zeile 7-9) die durchschnittliche Temperatur berechnet und an den `outputStream` weitergereicht.

Esper

In Esper können Windows auf die gleiche Art und Weise wie in Siddhi realisiert werden. Entweder werden Windows am Ereignistyp der Ereignisregel definiert, wie in der Fallstudie umgesetzt (siehe Abschnitt 4.4.2), oder sie lassen sich als Named Windows erstellen, sodass sie für weitere Ereignisregeln verwendbar sind. [Esp18f] Esper nutzt dafür die `CREATE`-Klausel, um Named Windows zu konstruieren.

Sliding Time Window

```
1 create window TempEventWindow#time(40 sec) as TempEvent
2
3 insert into TempEventWindow
4 select *
5 from TempEvent;
6
7 select avg(temp) as avgTemp
8 from TempEventWindow
```

Auflistung 5.5: Ereignisanfrage mit einem Sliding Time Window als Named Window in Esper

Das Code-Beispiel 5.5 zeigt ein Named Window als Sliding Time Window und wird analog zu Siddhi mit der `time()`-Funktion realisiert (Zeile 1). Damit bestimmt wird, welchen Ereignistyp das Sliding Time Window einspeisen kann, wird mithilfe des `AS`-Operators der Ereignistyp festgelegt. Alternativ auch mit `AS SELECT` und den Ereignisattributen, welche die eintreffenden Ereignisse spezifizieren. Das Code-Beispiel 5.5 setzt dabei die gleiche Ereignisregel wie in den vorherigen Siddhi Window-Beispielen um. Das Sliding Time Window enthält alle `TempEvents`, die innerhalb der letzten 40 Sekunden eintreffen. Anschließend wird mit der zweiten Ereignisanfrage (Zeile 7-8) für alle aktuellen `TempEvents` im Window die Durchschnittstemperatur ausgerechnet, welche ggf. von einem definierten Listener (ähnliche Funktion wie der Subscriber) ausgegeben werden kann, der zwischen aktuellen und abgelaufenen Ereignissen unterscheidet.

Batch Time Window

```
1 create window TempEventWindow#time_batch(40 sec) as TempEvent
2
3 insert into TempEventWindow
4 select *
5 from TempEvent;
6
7 select avg(temp) as avgTemp
8 from TempEventWindow
```

Auflistung 5.6: Ereignisanfrage mit einem Batch Time Window Named Window in Esper

Das Code-Beispiel 5.6 zeigt ein Batch (Tumbling) Time Window und wird mit `time_batch` und dem Ereignistyp `TempEvent` definiert (Zeile 1). Es verarbeitet alle `TempEvents` als Batch (Stapel), die jede 40 Sekunden eintreffen. Die zweite Ereignisanfrage (Zeile 7-8) berechnet ebenfalls die Durchschnittstemperatur der aktuellen `TempEvents`.

Da Esper die Sliding Length Windows und Batch Length Windows auf die gleiche Art und Weise wie die Sliding Time Windows und Batch Time Windows umsetzt, die bei Siddhi bereits erklärt wurden, wird auf weitere Beispiele verzichtet. Esper besitzt eine beachtliche Menge an Fensterarten mit zusätzlichen Konfigurationensparametern, die es im aktuellen Stand von Siddhi nicht gibt. [Esp18f]

5.1.3. Zeitkonzept

Mithilfe des Kriteriums *Zeitkonzept* sollen die möglichen Varianten zur Änderung des Zeitkonzepts der beiden Engines betrachtet und verglichen werden.

Siddhi

In Siddhi gibt es die Möglichkeit, die Systemzeit zu manipulieren, sodass die Systemzeit gemäß den Zeitstempeln der Ereignisse eingestellt ist. Siddhi verwendet intern die Klasse `TimestampGeneratorImpl` zur Erzeugung von Zeitstempeln. [Suh18] Standardmäßig erhalten eintreffende Ereignisse automatisch einen Zeitstempel basierend auf der aktuellen Systemzeit der JVM, auf der die Siddhi-Engine läuft. Dieses übliche Zeitkonzept kann umgangen werden, indem die `@app:playback`-Annotation zur Siddhi-Anwendung hinzugefügt wird.

```
1 @app:playback(idle.time = '500 millisecond', increment = '2 sec')
2
3 define stream TempEventStream(sensorID string, timestamp long, roomNu int,
      temp double);
4
5 from TempEventStream#window.timeBatch(1 sec)
6 select *
7 insert into OutputStream;
```

Auflistung 5.7: Zählt die Systemzeit hoch, um Ereignisse nach ihren angegebenen Zeitstempel zu empfangen [vgl. WSO18l]

Durch die `@app:playback`-Annotation nutzt die `TimestampGeneratorImpl`-Klasse eine Systemzeit basierend auf dem Zeitstempel des ersten eintreffenden Ereignisses, welcher als Ereignisattribut angegeben ist. [Suh18] Das Code-Beispiel 5.7 zeigt die zwei zusätzlichen Parameter, die zur Zeitkonfiguration der `@app:playback`-Annotation übergeben werden können. Mit `idle.time` kann eine Wartezeit für eingehende Ereignisse angegeben werden. Sollten in diesem Zeitintervall keine weiteren Ereignisse eintreffen, wird die Siddhi Systemzeit um die angegebene Anzahl an Sekunden des `increment`-Parameters erhöht. [WSO18a] Dies ist besonders bei Testszenarien in Siddhi-Anwendungen sehr hilfreich, um Zeitfenster auf korrekte Verarbeitung zu überprüfen. Das Code-Beispiel 5.7 zeigt eine mögliche Zeitmanipulation und erhöht jeweils die Systemzeit um zwei Sekunden, wenn im Zeitintervall von 500 Millisekunden keine `TempEvents` in das Batch Time Window eintreffen.

Esper

Die Esper-Engine kann so konfiguriert werden, dass die standardmäßige interne Engine-Zeit (JVM-Systemzeit) ausgeschaltet wird und eintreffende Ereignisse einen Zeitstempel einer externen bereitgestellten Zeit erhalten. Dies kann vor der Initialisierung der Engine mit der `Configuration`-Schnittstelle vorgenommen werden. [Esp18f]

```
1 Configuration config = new Configuration();
2 config.getEngineDefaults().getThreading().setInternalTimerEnabled(false);
3 EPServiceProvider engine = EPServiceProviderManager.getDefaultProvider("epa", config);
4
5 long milliseconds = System.currentTimeMillis();
6 CurrentTimeEvent timeEvent = new CurrentTimeEvent(milliseconds);
7 engine.getEPRuntime().sendEvent(timeEvent);
```

Auflistung 5.8: Deaktiviert interne Zeit der CEP Engine und liefert die aktuelle Systemzeit, indem ein `CurrentTimeEvent` registriert wird [vgl. Esp18f]

Das Code-Beispiel 5.8 zeigt dabei die Deaktivierung der internen Zeit eines EPA mit der `setInternalTimerEnabled()`-Methode, indem `false` übergeben wird (Zeile 2). Anschließend kann durch senden eines `CurrentTimerEvent` die aktuelle Systemzeit festgelegt

werden (Zeile 7). Alle eintreffenden Ereignisse bekommen einen Zeitstempel basierend ab der festgelegten Zeit des `CurrentTimeEvent`. [Esp18f]

```
1  EPServiceProvider engine = EPServiceProviderManager.getDefaultProvider("epa");
2  EPRuntime runtime = engine.getEPRuntime();
3
4  runtime.sendEvent(new TimerControlEvent(TimerControlEvent.ClockType.CLOCK_EXTERNAL));
5  runtime.sendEvent(new CurrentTimeEvent(0));
6
7  epaAdmin.createEPL("select * from TempEvent output every 2 seconds");
8
9  runtime.sendEvent(new CurrentTimeEvent(1000));
10 runtime.sendEvent(new CurrentTimeEvent(3000));
```

Auflistung 5.9: Schaltet auf externe Zeit um und sendet zur Voranschreitung der Zeit `CurrentTimeEvents` [vgl. Esp18f]

Eine weitere Alternative ist das Senden von `CurrentTimeEvents` zur Laufzeit. Dafür wird wie in Code-Beispiel 5.9 zu sehen, mithilfe des `TimerControlEvent` auf die externe Zeit umgeschaltet (Zeile 4). Anschließend wird die Zeit mit einem `CurrentTimeEvent` auf 0 gesetzt (Zeile 5) und im Weiteren durch gesendete `CurrentTimeEvents` vorangetrieben (Zeile 9-10). [Esp18f]

5.1.4. Datenextraktion & Umwandlung verschiedener Datenformate

Mithilfe der Kriterien *Datenextraktion & Umwandlung verschiedener Datenformate* soll untersucht werden, welche Möglichkeiten Siddhi und Esper bieten, um Ereignisse aus verschiedenen Quellen mit unterschiedlichen Datenformaten zu beziehen und bei Veröffentlichung wieder in ein entsprechendes Datenformat umzuwandeln.

Siddhi

In Siddhi-Anwendungen ist es mittels Source Mapper möglich, Ereignisse aus verschiedenen Quellen und in unterschiedlichen Datenformaten zu empfangen. In Abschnitt 4.3 der Implementierung der Fallstudie wurde bereits das Auslesen von Ereignissen aus einer CSV-Datei vorgestellt, welches mit einer Konfiguration der `@Source`-Annotation und der `@Map`-Annotation realisiert wurde. Durchaus sind mit den beiden Annotationen auch weitere Quelltypen und Datenformate möglich.

```
@Source(type = 'websocket', url = 'ws://localhost:8025/websockets/temp',
@Map(type='xml'))
define stream TempEventStream (sensorID string, roomNu int, temp double);
```

Auflistung 5.10: Empfangen von XML-Ereignissen über ein WebSocket in Siddhi [vgl. WSO18h]

Das Code-Beispiel 5.10 zeigt einen `TempEventStream`, der Ereignisse über einen integrierten WebSocket-Adapter empfängt, die im XML-Datenformat vorliegen. Die `@Source`-Annotation legt dementsprechend mit dem Parameter den Quelltyp und die URL des entfernten Endpunktes fest. Die `@Map`-Annotation bestimmt anschließend den Datentyp, in dem die Ereignisse eintreffen. Die eintreffenden XML-Ereignisse des WebSocket-Servers werden in das Siddhi-Ereignisformat konvertiert und im `TempEventStream` zur weiteren Verarbeitung hinzugefügt. [WSO18h]

```
@Sink(type = 'websocket', url = 'ws://localhost:8025/temp',
@Map(type='xml'))
define stream TempEventOutputStream (sensorID string, roomNu int, temp
    double);
```

Auflistung 5.11: Veröffentlichung von XML-Ereignissen über ein WebSocket in Siddhi [vgl. WSO18h]

Sind in Siddhi Ereignisse verarbeitet worden, können diese an verschiedene Ereignissenken und in unterschiedlichen Datenformaten mittels Sink Mappers veröffentlicht werden. In der Fallstudienumsetzung (siehe Abschnitt 4.3) wurden die Siddhi-Ereignisse an Ereignisströme weiterer Siddhi EPA-Instanzen verschickt, die innerhalb derselben JVM liegen. Durchaus ist das Senden von Ereignissen an weitere Senktypen möglich. Das Code-Beispiel 5.11 zeigt dabei das Gegenbeispiel des eben vorgestellten Source Mappers. Alle eintreffenden Ereignisse des `TempEventOutputStream` werden über ein WebSocket-Adapter an die angegebene URL `ws://localhost:8025/temp` gesendet.

Wird beim Source Mapper oder Sink Mapper keine `@Map`-Annotation übergeben, wird der Standardquelltyp bzw. Standardsenktyp `passThrough` genutzt. Der Source Mapper geht folgend davon aus, dass die Ereignisse im Siddhi-Ereignisformat vorliegen. Der Sink Mapper hingegen veröffentlicht die Ereignisse im Siddhi-Ereignisformat an die Ereignissenke. Aus diesem Grund ist es wichtig, darauf zu achten, dass die richtigen Typen in der `@Source`-Annotation sowie `@Sink`-Annotation übergeben werden. [WSO18h]

Zusätzlich ist es möglich, durch *Distributed Sinks* (verteilte Senken) die Ereignisse an mehrere Endpunkte zu veröffentlichen. So lässt sich mit einer Konfiguration das gemeinsame Mapping der Ereignisse für alle Endpunkte mit optionalen Verteilungsstrategien wie z. B. Round Robin definieren. Ein Code-Beispiel befindet sich in der Siddhi Dokumentation und wird an dieser Stelle nicht weiter aufgegriffen. [WSO18l]

Weitere integrierte Adapter für Siddhi sind z. B. HTTP, Email, RabbitMQ, JMS oder Amazon Simple Queue Service (SQS). [WSO18f]

Esper

Wie bereits in Abschnitt 2.5 erwähnt, unterstützt Esper selbst einige native Datenformate bei Ein- und Ausgabeereignissen wie Java-Objekte (POJO, Java Bean), Maps, Objekt-Arrays und XML-Dokumente. [Esp18g]

```
1 JSONEventRenderer renderJson = epa.getEPRuntime().getEventRenderer().getJSONRenderer(stmt.
      getEventType());
2 String jsonEvent = renderJson.render("TempEvent", event);
```

Auflistung 5.12: Umwandlung der Ereignisse in JSON [vgl. Esp18f]

Liegen beispielsweise Ereignisse als Java-Objekt vor, kann Esper diese zur Ausgabe mithilfe der `EventRenderer`-Klasse in XML oder JSON umwandeln. Das Code-Beispiel 5.12 zeigt den `JSONEventRenderer`, der jedes Ereignis des Ereignistyps `TempEvent` in JSON rendert. [Esp18g]

```
1  Configuration config = new Configuration();
2
3  String httpConfig = "<esperio-http-configuration>\n" +
4                      "<service name=\"httpService\" port=\"8025\"/>" +
5                      "<get service=\"httpService\" pattern=\"*\"/>" +
6                      "</esperio-http-configuration>";
7
8  config.addPluginLoader("EsperIOHTTPAdapter", EsperIOHTTPAdapterPlugin.class.getName(), new
       Properties(), httpConfig);
9
10 EPServiceProvider epa = EPServiceProviderManager.getProvider("epa", config);
11
12 ConfigurationHTTPAdapter adaptConfig = new ConfigurationHTTPAdapter();
13 Request request = new Request();
14 request.setStream("AlertEvent");
15 request.setUri("http://localhost:80/root/event");
16 adaptConfig.getRequests().add(request);
17
18 EsperIOHTTPAdapter httpOutputAdapter = new EsperIOHTTPAdapter(adaptConfig, "epa");
19
20 httpOutputAdapter.start();
```

Auflistung 5.13: Ereignistransport über HTTP-Adapter in Esper [vgl. Esp18f]

In Abschnitt 4.4.1 wurde bereits der Einsatz des `CSVInputAdapter` gezeigt, um Ereignisse aus einer CSV-Datei zu lesen. Esper bietet neben dem `CSVInputAdapter` noch weitere Adapter für z. B. HTTP, Spring JMS, Sockets oder Advanced Message Queuing Protocol (AMQP) an, die je nach Zweck, über die dazugehörige API konfiguriert werden können. [Esp18g] Das Code-Beispiel 5.13 soll eine demonstrative Umsetzung mit dem HTTP-Adapter zeigen.

Um den HTTP-Adapter zu verwenden, müssen zunächst einmal die entsprechenden Softwareabhängigkeiten in die pom.xml des Maven-Projekts oder die JAR-Dateien `esperio-http-version.jar, httpcore-version.jar` und `httpclient-version.jar` dem Klassenpfad beigefügt werden. [Esp18g]

Der HTTP-Dienst, der die HTTP-Verbindung festlegt, kann als XML-String definiert werden. Der HTTP-Dienst wird entsprechend über den Namen `httpService` und den Port `8025` verfügbar gemacht. Damit ein oder mehrere GET-Handler sich beim HTTP-Dienst registrieren können, geschieht dies ebenfalls in der Konfiguration mit `get service = "httpService"` (Zeile 5). [Esp18g] Werden Ereignisse über eine URI an den HTTP-Eingangsadapter geschickt, muss die GET-Request URI den Ereignistyp sowie weitere Ereignisattribute enthalten, z. B. `http://localhost:8025/sendevent?Stream=TempEvent&sensorID=sensor1&roomNu=2&temp=21.6` für ein `TempEvent` mit den entsprechenden Attributen `sensorID, roomNu` und `temp`.

Ebenso lassen sich verarbeitete Ereignisse als URI über den HTTP-Ausgabeadapter verschicken. Das `Request`-Objekt legt die URI und den entsprechenden Ereignistyp des Ausgangsstroms fest (Zeile 13-15). Es lassen sich auch weitere `Request`-Objekte anlegen, beispielsweise für jeden Ereignistyp mit entsprechender URI. Das `ConfigurationHTTPAdapter`-Objekt bekommt anschließend den angelegten `Request` hinzugefügt (Zeile 16). Zum Schluss wird die HTTP-Konfiguration `adaptConfig` und die URI der EPA-Instanz `epa` bei Erzeugung des `EsperIOHTTPAdapters` mit übergeben (Zeile 18), welcher daraufhin gestartet werden kann (Zeile 20). Der Ausgangsstrom löst, sobald die verarbeiteten Ereignisse eintreffen, den GET-Handler aus. Basierend auf den `Request`-Konfigurationen werden beispielsweise `AlertEvents` mit den resultierenden Ereignisattributen als folgende URI `http://localhost:80/root/event?stream=AlertEvent&sensorID=sensor4&roomNu=3&avgTemp=15.5` gesendet. [Esp18g]

5.1.5. Ereignispersistenz

Das Kriterium *Ereignispersistenz* soll zeigen, welche Möglichkeiten die beiden Engines bieten, um verarbeitete Ereignisse mit ihren enthaltenen Daten zu speichern.

Siddhi

Üblicherweise werden in Siddhi Ereignisse für 500 Millisekunden im Arbeitsspeicher bereitgehalten und wieder gelöscht. Mithilfe von angelegten Ereignistabellen können Ereignisse dauerhaft gespeichert und für weitere Siddhi Ereignisanfragen verwendet werden. [WSO18l]

```
1 @Index('roomNu')
2 @PrimaryKey('sensorID')
3 define table TempEventTable (sensorID string, roomNu int, temp double);
4
5 from TempEventStream
6 select *
7 insert into TempEventTable;
```

Auflistung 5.14: Erstellung einer Ereignistabelle in Siddhi

Die Erstellung der Ereignistabelle `TempEventTable` in Code-Beispiel 5.14 (Zeile 1-3) ist dabei ähnlich zu der von Ereignisströmen. Das Code-Beispiel 5.14 legt durch die Verwendung von `define table` die Ereignistabelle `TempEventTable` mit den Tabellenspalten `sensorID` als String, `roomNu` als Integer und `temp` als Double im Arbeitsspeicher (In-Memory) an, falls diese noch nicht existiert. Mit der `@PrimaryKey`-Annotation wird der eindeutige Primärschlüssel der Tabelle definiert, d. h. analog zu einer Datenbanktabelle muss jeder Eintrag der Ereignistabelle gemäß des Primärschlüssels eindeutig sein. In Code-Beispiel 5.14 wurde als Primärschlüssel die `sensorID` gewählt (Zeile 2), es können jedoch auch weitere Tabellenspalten angegeben werden. Des Weiteren lassen sich auch Tabellenspalten als Indizes festlegen, indem sie der `@Index`-Annotation übergeben werden (Zeile 1), um einen schnelleren Zugriff auf die Daten der Ereignistabelle zu gewährleisten. Auch hier ist es möglich, mehrere Tabellenspalten zu übergeben. [WSO18l]

```
1 @Store(type='cassandra', column.family='TempEventTable',
2        keyspace='cassandraTable', cassandra.host='localhost')
3 @Index('roomNu')
4 @PrimaryKey('sensorID')
5 define table TempEventTable (sensorID string, roomNu int, temp double);
6
7 from TempEventStream
8 select *
9 insert into TempEventTable;
```

Auflistung 5.15: Ereignisspeicherung in eine Cassandra-DB-Instanz

Durch die Siddhi-Speichererweiterungen können die Ereignistabellen auch in Datenbanken wie z. B. Cassandra oder MongoDB angelegt werden.

Das Code-Beispiel 5.15 erstellt durch die Verwendung der `@Store`-Annotation eine Ereignistabelle `TempEventTable` ebenfalls mit den Tabellenspalten `sensorID` als String, `roomNu` als Integer und `temp` als Double in der Cassandra-Instanz, falls diese noch nicht existiert. Die weiteren Verbindungsanweisungen für diese Ereignistabelle mit der Cassandra-Instanz werden in der `@Store`-Annotation konfiguriert. [WSO18k]

Für die Ereignistabellen lassen sich auch spezifische Ereignisanfragen mit `DELETE`, `UPDATE` und `JOIN` durchführen. [WSO18l]

Esper

```
1 create table TempEventTable(sensorID string primary key, roomNu int, temp
      double)
2
3 insert into TempEventTable
4 select *
5 from TempEventStream
```

Auflistung 5.16: Ereignisspeicherung in Esper mithilfe von Tabellen

Wie das Code-Beispiel 5.16 zeigt, werden in Esper die Ereignistabellen mit `create table` erstellt (Zeile 1). Um den Primärschlüssel festzulegen, wird dieser direkt in der Definition der Tabellenspalten (`sensorID, roomNu` und `temp`) gesetzt. In Code-Beispiel 5.16 wird dafür nach `sensorID` das Schlüsselwort `primary key` verwendet (Zeile 1). [Esp18f]

```
1 ConfigurationDBAdapter adapterConfig = new ConfigurationDBAdapter();
2 ConfigurationDBRef dbConfig = new ConfigurationDBRef();
3 dbConfig.setDriverManagerConnection("com.mysql.jdbc.Driver", "jdbc:mysql://localhost/testdb"
      , "user" ,"password");
4 adapterConfig.getJdbcConnections().put("database1", dbConfig);
5
6 EsperIODBAdapter dbAdapter = new EsperIODBAdapter(adapterConfig, "epa");
7 dbAdapter.start();
```

Auflistung 5.17: Ereignisspeicherung in relationaler Datenbank mit JDBC-Verbindung in Esper [vgl. Esp18f]

Um Ereignisse ebenfalls in Datenbanken zu speichern, kann dies mit dem EsperIO-Datenbankadapter erfolgen. Dafür muss lediglich der JDBC-Driver und die entsprechende Softwareabhängigkeit `esperio-db-version` in der pom.xml des Maven-Projekts oder als JAR-Datei eingefügt werden. So lässt sich beispielsweise eine JDBC-Verbindung durch entsprechende Konfigurationen des Adapters anhand der `ConfigurationDBAdapter`-Klasse vornehmen. Das Code-Beispiel 5.17 zeigt eine Adapterkonfiguration zur Datenbankverbindung (Zeile 1-4) und startet den Adapter über die API (Zeile 7). Die EPA-Instanz mit der URI `epa` kann anschließend den Datenbanknamen `database1` in den Ereignisanfragen verwenden, um mit der Datenbank zu kommunizieren. [Esp18f]

Esper bietet sogar die Möglichkeit für Tabellenspalten jeweils Aggregationsfunktionen zu definieren, um beispielsweise den Durchschnitt von Double-Werten in einer Ereignistabelle zu errechnen. Außerdem können auch Ereignistypen als Attributstyp der Tabellenspalte mithilfe der `@Type`-Annotation verwendet werden, um komplette Ereignisse in einer Spalte zu speichern. Aber auch Windows können in den Tabellendefinitionen als Spalte genutzt werden, was in Siddhi nicht möglich ist.

5.1.6. Behandlung von Ereignisduplikaten

Das Kriterium *Behandlung von Ereignisduplikaten* soll untersuchen, welche Möglichkeiten Siddhi und Esper zur Behandlung von Ereignisduplikaten bereitstellen.

Siddhi

Mit der Erweiterung *siddhi-execution-unique* lassen sich Ereignisduplikate umgehen, sodass nur eindeutige Ereignisse in den Ereignisströmen gefunden und verarbeitet werden. Dazu werden verschiedene Arten von Windows genutzt, damit eindeutige Ereignisse nach einem festgelegten Parameter `unique.key` aufgenommen werden. [WSO18e] Im Folgenden werden zwei Beispiele mit jeweils unterschiedlicher Art von Windows vorgestellt.

```
1 from TempEventStream#window.unique:first(sensorID, roomNu)
2 select *
3 insert into UniqueTempEventStream;
```

Auflistung 5.18: Verarbeitung von eindeutigen Ereignissen in Siddhi mithilfe eines `unique:first`-Windows

Das Code-Beispiel 5.18 zeigt ein `unique:first`-Windows, welches die ersten `TempEvents`, die gemäß den Schlüsselparametern `sensorID` und `roomNu` eindeutig sind, aufnimmt und in den `UniqueTempEventStream` hinzufügt. Alle `TempEvents` im Window haben daher eindeutige Werte für die Ereignisattribute, die als Schlüsselparameter gesetzt sind. Trifft anschließend ein weiteres `TempEvent` ein, welches einem im Window befindlichen `TempEvent` mit den angegebenen Schlüsselparametern nach gleicht, wird dieses Ereignis nicht im Window aufgenommen. [WSO18e]

```
1 from TempEventStream#window.unique:ever(temp)
2 select *
3 insert all events into UniqueTempEventStream;
```

Auflistung 5.19: Verarbeitung von eindeutigen Ereignissen in Siddhi mithilfe eines `unique:ever`-Windows

Das weitere Code-Beispiel 5.19 speist eindeutige `TempEvents` nach dem Schlüsselparameter `temp` ein (Zeile 1). Anders als beim ersten Beispiel wird ein eindeutiges `TempEvent` im Window aktualisiert, sobald ein äquivalentes `TempEvent` eintrifft, d. h. das vorhandene `TempEvent` im Window läuft ab und wird durch das neue `TempEvent` mit dem gleichen Wert des eindeutigen Schlüsselparameters `temp` ersetzt.

Esper

In Esper kann zur Erkennung von Ereignisduplikaten entweder das `firstUnique`-Window oder die `PATTERN`-Klausel mit `every-distinct` verwendet werden. [Esp18f]

```
1 select *
2 from TempEvent#firstunique(sensorID, roomNu) --alternativ #time(20)
    hinzufuegen
```

Auflistung 5.20: Verarbeitung von eindeutigen Ereignissen in Esper mithilfe eines `firstUnique`-Windows

Das erste Code-Beispiel 5.20 zeigt das `firstUnique`-Window, welches nur das erste `TempEvent` gemäß den eindeutigen Schlüsselparametern `sensorID` und `roomNu` einspeist (Zeile 2). Weitere eintreffende `TempEvents` mit der gleichen `sensorID` werden ignoriert. Alternativ lässt sich auch ein weiteres Zeit- oder Längenfenster hinzufügen, wenn nur Duplikate auf bestimmte Zeit oder nach Anzahl ignoriert werden sollen. [Esp18f]

```
1 select *
2 from pattern[every-distinct(t.temp) t=TempEvent] -- alternativ mit where
    timer:within(20 sec)
```

Auflistung 5.21: Verarbeitung von eindeutigen Ereignissen in Siddhi mithilfe eines `unique:ever`-Windows

Das Code-Beispiel 5.21 zeigt die Realisierung mit der `PATTERN`-Klausel und dem Schlüsselwort `every-distinct`. Es werden jeweils die ersten eintreffenden `TempEvents` für den Schlüsselparameter `sensorID` eingespeist. Auch hier ist eine Längen- oder Zeitbegrenzung möglich, um Ereignisduplikate zu ignorieren, z. B. mit `where timer:within(10 sec)`. [Esp18f]

5.1.7. Parallelisierung von Ereignisverarbeitung (Multithreading)

Dieses Kriterium *Parallelisierung von Ereignisverarbeitung* soll zeigen, ob beide Engines Multithreading zur Parallelisierung von Ereignisanfragen bieten.

Siddhi

In Abschnitt 3.4 wurde erwähnt, dass sich die Ereignisverarbeitung in Siddhi parallelisieren lässt. Bedauerlicherweise gibt es nur geringe Informationen in Siddhis Dokumentation, um einen wichtigen Aufschluss über die asynchrone Verarbeitung durch Threading

zu erhalten. Aus diesem Grund musste sich teilweise auf einem Beitrag eines WSO2 Software Entwicklers [Dah18] auf StackOverflow gestützt werden.

Üblicherweise puffert Siddhi Ereignisse vor der Verarbeitung nicht und der gesamte Ablauf der Siddhi-Anwendung verhält sich synchron, d. h. die gesamte Ereignisverarbeitung läuft innerhalb eines Threads ab. [Dah18] Mit der `@Async`-Annotation können jedoch Ereignisse, die in einem Ereignisstrom eintreffen, vorher gepuffert und anschließend von einem gesonderten Thread aufgenommen und weiterverarbeitet werden. [WSO18d] Siddhi verwendet dafür die frei verfügbare Bibliothek *Disruptor*[3], um den Verarbeitungsfluss asynchron zu machen. So lassen sich mehrere parallele Threads für dieselbe Verarbeitungslogik ausführen, die aber jeweils ein anderes Ereignis weiterleiten (Multithreading). [Dah18]

```
1 @async(buffer.size='512', workers='2', batch.size.max='20')
2 define stream TempEventStream(sensorID string, roomNu int, temp double);
```

Auflistung 5.22: Multithreading-Beispiel in Siddhi mithilfe der `@Async`-Annotation [vgl. WSO18l]

In der `@Async`-Annotation können entsprechende Konfigurationsparameter übergeben werden, wie in Code-Beispiel 5.22 dargestellt. So lässt sich mit `buffer.size` die Größe des Ereignispuffers festlegen, der an weitere Threads zur Weiterverarbeitung übergeben wird (ist standardmäßig auf 1024 gesetzt, wenn nicht anders definiert). Der `workers`-Parameter gibt die Anzahl der Threads an, die zur Verarbeitung der gepufferten Ereignisse verwendet werden (ist standardmäßig auf 1 gesetzt, wenn nicht anders definiert). Der letzte Parameter `batch.size.max` gibt die maximale Anzahl von Ereignissen an, die von einem Thread gemeinsam verarbeitet werden (ist standardmäßig auf die Größe von `buffer.size` gesetzt, wenn nicht anders definiert). [WSO18l]

Esper

Die Ereignisverarbeitung in einer Esper-Anwendung kann sich ebenfalls Multihreading verhalten. Auch in Esper wird die Ereignisverarbeitung innerhalb eines Threads (Anwendungsthread) durchlaufen. Dieser Thread wird kurzzeitig blockiert, bis Ereignisse von einem `Listener` oder `Subscriber` als Ausgangsereignisse verarbeitet wurden, z. B. Schreiboperationen in eine relationale Datenbank. Daher ist es hilfreich, die Ereignisse asynchron zu verarbeiten und bestimmte Operationen in weitere Threads auszulagern, um Input- oder Output-Bottlenecks zu vermeiden. Dazu können in Esper verschiedene Threading-Varianten verwendet werden. [Esp18f]

[3] Disruptor ist eine Java-Bibliothek, die in asynchronen EDAs verwendet wird. Die Bibliothek nutzt dafür eine array-basierte zirkulare Datenstruktur (ring buffer) und reduziert dabei Schreibkonflikte zwischen Threads. [Bae18]

```
1 Configuration config = new Configuration();
2 config.getEngineDefaults().getThreading().setThreadPoolInbound(true);
3 config.getEngineDefaults().getThreading().setThreadPoolInboundNumThreads(3);
```

Auflistung 5.23: Multithreading-Beispiel mithilfe der Inbound-Threading-Variante in Esper [vgl. Esp18f]

Das Code-Beispiel 5.23 zeigt die Inbound-Threading-Variante, die über die `Configuration`-API aktiviert wird (Zeile 2). Mit der `setThreadPoolInboundNumThreads`-Methode kann die Anzahl an Threads angegeben werden, die sich im Thread-Pool befinden sollen (Zeile 3). Bei Ausführung der Ereignisverarbeitung platziert die Engine die eintreffenden Ereignisse in eine Warteschlange. Alle Threads, die sich im Thread-Pool befinden, können auf die Warteschlange zugreifen und die Ereignisse weiter verarbeiten und übermitteln. Der Anwendungsthread wird daher nach senden eines Ereignisses über die `sendEvent`-Methode nicht weiter blockiert und kann weiter fortfahren. [Esp18f]

In Esper ist es auch möglich die Ereigniskapazität der Warteschlangen zu bestimmen oder Locks für Threads zu konfigurieren, auf die aber an dieser Stelle nicht weiter eingegangen wird. [Esp18f]

5.2. Technischer Vergleich

5.2.1. Skalierbarkeit

Mit dem Kriterium *Skalierbarkeit* soll untersucht werden, inwiefern sich die beiden Engines horizontal skalieren lassen, um die Leistung der Ereignisverarbeitung zu steigern.

Siddhi

In Abschnitt 4.3 wurde gezeigt, wie die EPA-Instanzen untereinander kommunizieren können, wenn diese in der gleichen JVM existieren, aber in verschiedenen Anwendungslaufzeiten ausgeführt werden. Sind die EPA-Instanzen in Siddhi auf weitere Rechnerknoten aufgeteilt, so ergibt sich daraus, wie in Abschnitt 2.4.2 erwähnt, ein EPN. Innerhalb dieses EPN müssen die EPA-Instanzen miteinander kommunizieren, um Ereignisse auszutauschen und weiterzuverarbeiten. In Abschnitt 2.4.1 erläutert, lässt sich zur Kommunikation beispielsweise eine MOM mit dem Publish&Subscribe-Prinzip einsetzen. Das Code-Beispiel 5.24 und Code-Beispiel 5.25 zeigen jeweils demonstrativ die Kommunikation über einen RabbitMQ Message Broker.

```
define stream TempEventStream(sensorID string, roomNu int, temp double);
@sink(type ='rabbitmq', uri = 'amqp://user:password@localhost:8025',
exchange.name = 'direct', routing.key= 'direct', @map(type='xml'))
define stream AlertEventStreams(sensorID string, roomNu int, avgTemp
    double);

from TempEventStream
select sensorID, roomNu, avg(temp) < 12.5
insert into AlertEventStream;
```

Auflistung 5.24: Versenden der Ereignisse durch RabbitMQ-Sink in Siddhi

In Code-Beispiel 5.24 werden zunächst die verarbeiteten Ereignisse des `TempEventStreams` in den Ausgangsstrom `AlertEventStream` hinzugefügt. Dieser Ausgangsstrom besitzt die `@Sink`-Annotation, um die Ereignisse mithilfe des AMQP-Protokolls an den RabbitMQ-Broker für die Ziel-Instanz zu veröffentlichen, die sich auf einem anderen Rechnerknoten befindet. [WSO18g]

```
@source(type ='rabbitmq', uri = 'amqp://user:password@localhost:8025',
exchange.name = 'direct', routing.key= 'direct', @map(type='xml'))
define stream AlertEventStreams(sensorID string, roomNu int, avgTemp
    double);
```

Auflistung 5.25: Empfangen der Ereignisse durch RabbitMQ-Source in Siddhi

Im Gegenzug enthält die Ziel-Instanz in Code-Beispiels 5.25 die `@Source`-Annotation und implementiert den Ausgangsstrom `TempEventStream` als ihren Eingangsstrom, um die eintreffenden Ereignisse vom RabbitMQ-Broker über das AMQP-Protokoll zu empfangen. Beide Annotationen bekommen als Parameter die identische URI für die Verbindung zum RabbitMQ-Server sowie den `exchange.name` und `routing.key`, um den Austausch der Ereignisse zu bestimmen, wie diese an Warteschlangen weitergereicht werden. [WSO18g] Dieses Prinzip ist für weitere EPA-Instanzen eines Clusters in Siddhi möglich und kann mit verschiedenen Kommunikationsprotokollen der Siddhi-Erweiterungen umgesetzt werden.

Esper

Esper lässt sich ebenfalls horizontal skalieren, indem weitere EPAs auf mehrere Rechnerknoten verteilt werden und dementsprechend als Cluster fungieren. Jeder Rechner kann ein oder mehrere EPA-Instanzen besitzen, die mit weiteren EPAs auf anderen Rechnern durch Ereignisaustausch kommunizieren und sich dadurch ebenfalls ein EPN (siehe Abschnitt 2.4.2) ergibt. [BD10]

```
1 public class TempEventSubscriber {
2     public void update(TempEvent event) {
3         EPServiceProvider epa2 = EPServiceProvicerManager.getProvider("epa2");
4         EPRuntime epa2Runtime = epa2.getEPRuntime();
5         epa2Runtime.sendEvent(event);
6     }
7 }
```

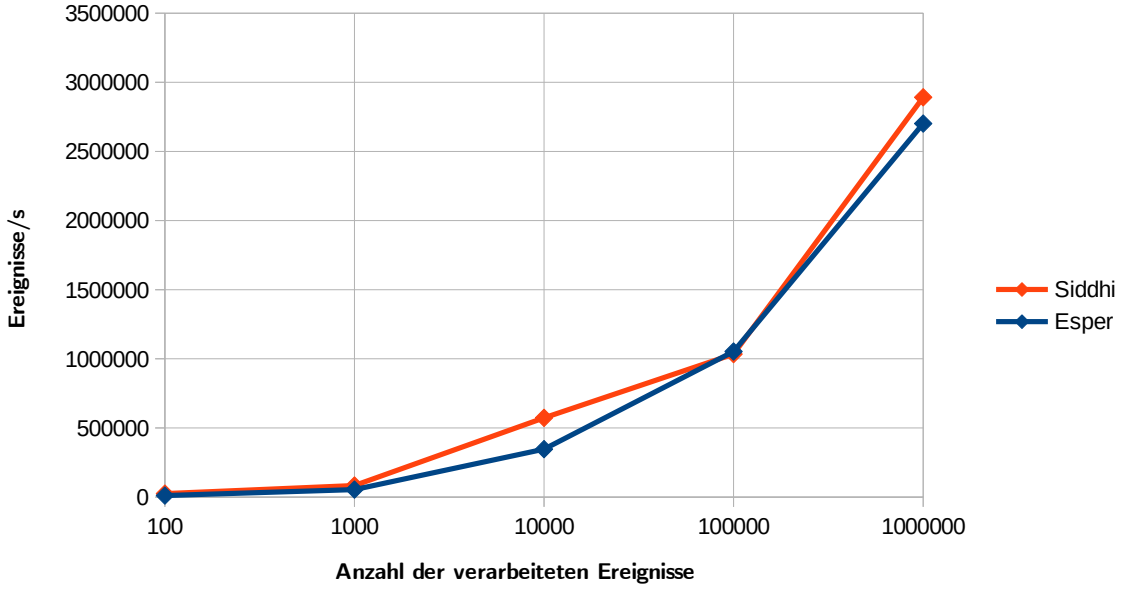
Auflistung 5.26: Versenden der Ereignisse an weitere Ziel-EPAs durch Subscriber-Klasse in Esper

Sobald eine EPA-Instanz durch `EPServiceProviderManager.getProvider(„epa“)` (siehe Abschnitt 4.4.1) erzeugt wurde, lassen sich die Ausgangsereignisse der EPA-Instanz anhand des registrierten Subscriber-Objekts an weitere EPAs weiterleiten. Das Code-Beispiel 5.26 zeigt die entsprechende Subscriber-Klasse mit der enthaltenen `update`-Methode. Innerhalb der `update`-Methode wird sowohl die Referenz des Ziel-EPAs mit `getProvider(„epa2“)` beschafft als auch das dazugehörige Laufzeitobjekt mit `getEPRuntime()` (Zeile 3-4). Anschließend werden die `TempEvents`, die der `update`-Methode übergeben werden, an den Ziel-EPA mit `send()` geschickt (Zeile 5). [BD10]

5.2.2. Performanz

Mit dem Kriterium *Performanz* soll die Leistungsfähigkeit der beiden Engines verglichen werden. Dabei wurde für jede Engine ein Performanztest durchgeführt. Um Siddhi und Esper empirisch zu bewerten, wurden die beiden CEP Engines unter den gleichen Bedingungen getestet. Der Performanztest soll den Durchsatz der verarbeiteten Ereignisse pro Sekunde ermitteln. Der Test wurde mit drei Ereignisanfragen durchgeführt, die jeweils wichtige CEP Funktionen wie Filterung, Mustererkennung und Aggregation realisieren. Die Umsetzung der Regeln wurden für Siddhi in Abschnitt 4.3.2 und Esper in Abschnitt 4.4.2 bereits erklärt. Für jede Ereignisanfrage wurden die Ereignisse ohne Verzögerung über das System gesendet. Dabei wurden für 100, 1.000, 10.000, 100.000 und 1.000.000 Ereignisse, die die Bedingung in den Ereignisanfragen erfüllen, jeweils der Durchsatz pro Sekunde berechnet. Fairerweise wurden für jede Engine die gleichen Testdaten verwendet. Alle drei Ereignisanfragen wurden für beide Engines jeweils drei mal durchgeführt, sodass aus den resultierenden Einzelergebnissen der Durchschnitt als Endergebnis berechnet wurde. Die Testszenarien der Engines wurden auf einem Rechner durchgeführt, der über einen Intel Core i7-7700 Prozessor mit 3.60 GHz, 8 Kerne, 16 GB RAM und als Betriebssystem Ubuntu 16.04 LTS verfügt. Im Folgenden soll das Ergebnis des Performanztests für Siddhi und Esper gezeigt und diskutiert werden.

Ergebnis der 4. Ereignisregel

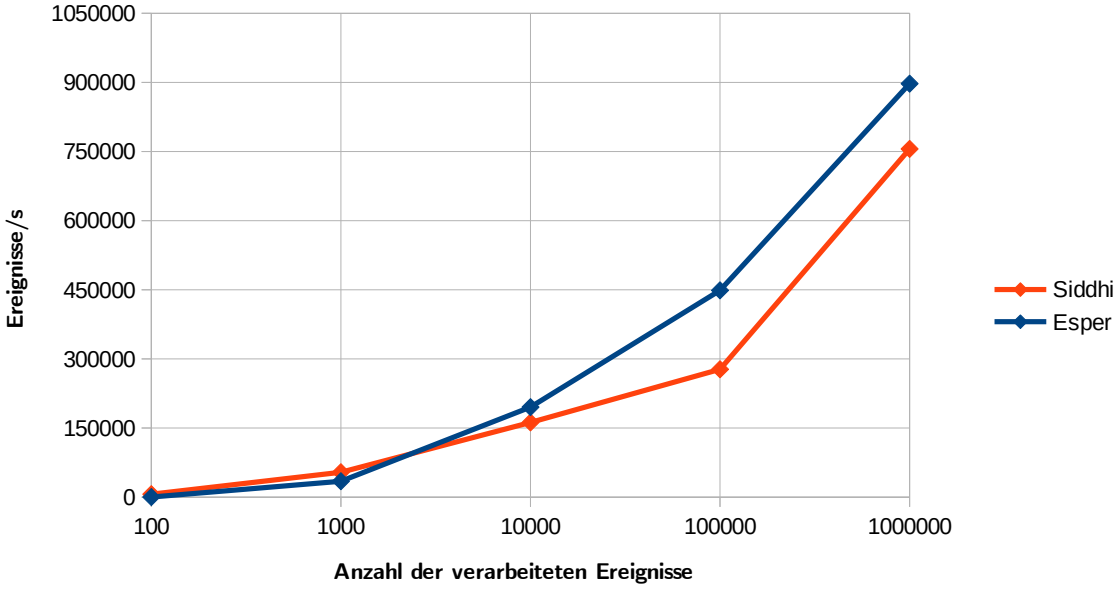


	100	1.000	10.000	100.000	1.000.000
Siddhi	25.252	83.496	573.098	1.033.767	2.891.392
Esper	11.203	53.999	346.500	1.052.463	2.701.414

Abbildung 5.1.: Das Durchsatzergebnis der beiden CEP Engines bei einer einfachen Filteranfrage.

Bei der einfachen Filteranfrage zeigen sich beide CEP Engines durchaus leistungsstark. Wie in Abbildung 5.1 zu sehen, verarbeitet Siddhi im Durchschnitt mehr Ereignisse pro Sekunde als Esper. Die CEP Engine Esper hält sich jedoch nur knapp schlechter als Siddhi und verarbeitet an der 100.000-Grenze kurzzeitig mehr Ereignisse pro Sekunde als Siddhi.

Ergebnis der 5. Ereignisregel

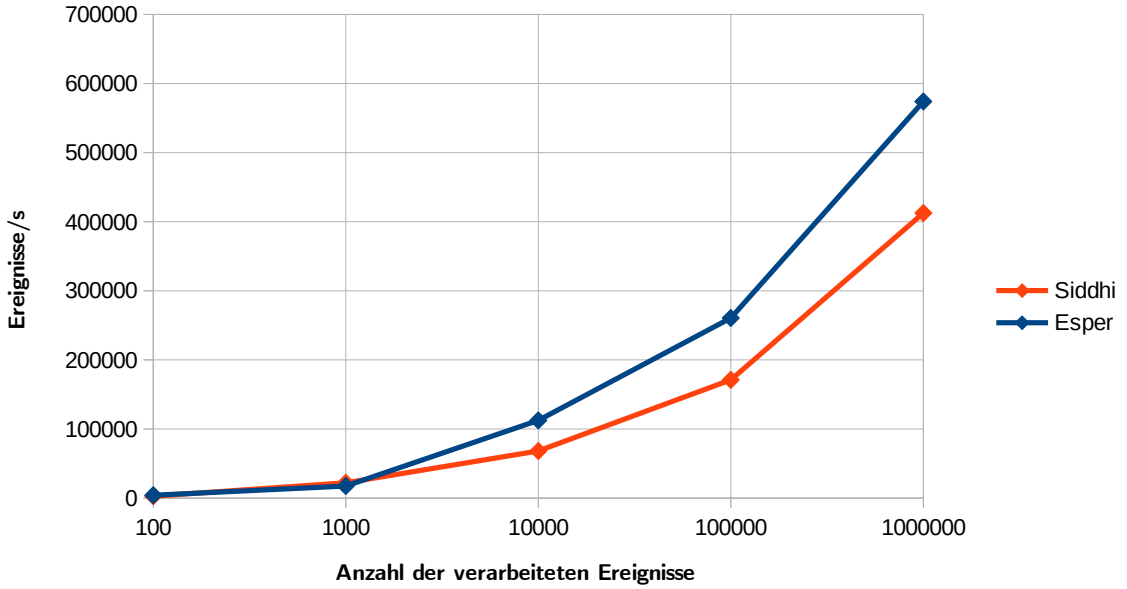


	100	1.000	10.000	100.000	1.000.000
Siddhi	6.388	53.999	162.148	277.396	755.688
Esper	242	34.502	195.297	448.788	897.255

Abbildung 5.2.: Das Durchsatzergebnis der beiden CEP Engines bei einer Aggregationsanfrage mit Längenfenster.

Bei der Aggregationsanfrage mit einem Längenfenster liegt die CEP Engine Esper vorne. Allerdings zeigt sich Siddhi anfangs besser und verarbeitet im Vergleich zu Esper überaus mehr Ereignisse pro Sekunde, welches im Liniendiagramm in Abbildung 5.2 zu erkennen ist. Siddhi bekommt jedoch ab der 1.000-Grenze Schwierigkeiten und überlässt Esper den Vorrang.

Ergebnis der 6. Ereignisregel



	100	1.000	10.000	100.000	1.000.000
Siddhi	2.799	22.098	68.169	171.090	412.526
Esper	4.181	17.490	112.471	260.578	573.979

Abbildung 5.3.: Das Durchsatzergebnis der beiden CEP Engines bei einer Musteranfrage

Der Performanztest für die Musteranfrage zeigt sich in Abbildung 5.3 für Siddhi im Allgemeinen schlechter. Siddhi verarbeitet ab der 1.000-Grenze durchaus weniger Ereignisse als Esper.

Grundsätzlich zeigen beide CEP Engines eine starke Leistung, haben jedoch je nach Art der Ereignisanfrage ihre Schwächen. Die CEP Engine Siddhi zeigt sich bei der 4. und 5. Ereignisregel anfangs stark, verarbeitet aber auf längerer Sicht weniger Ereignisse als Esper. Daher erweist sich Esper im Vergleich zu Siddhi als leistungsfähiger.

An diesem Punkt soll nochmal erwähnt werden, dass sich die Verarbeitung der Ereignisse auch weiter skalieren und parallelisieren lässt, wie es in Abschnitt 5.2.1 und Abschnitt 5.1.7 beschrieben wurde. In diesem Performanztest wurde dies jedoch nicht in Betracht gezogen, um den Test möglichst einfach zu halten. Dennoch kann durch die Verwendung von Parallelisierung und Skalierung einiges an Performanz dazu gewonnenen werden, erfordert jedoch auch den Einsatz performanter und zuverlässiger Hardware.

5.2.3. Fehlertoleranz

Das Kriterium *Fehlertoleranz* soll die Möglichkeiten der beiden Engines zeigen, welche bei Abbruch der Ereignisverarbeitung eingreifen, z. B. Wiederherstellungsmechanismus.

Siddhi

Sollte es während der Ereignisverarbeitung zu einem Ausfall oder sonstigen Problemen kommen, können in Siddhi mithilfe von periodischen *Snapshots* alle Zustandsinformationen und Windows in einem skalierbaren `PersistenceStore` gespeichert werden. So lässt sich nach Teilausfällen des Systems der letzte Zustand der Ereignisverarbeitung durch den letzten verfügbaren Snapshot wiederherstellen. [WSO18j] Folgendes Code-Beispiel 5.27 zeigt, wie sich der Verarbeitungszustand einer Siddhi-Anwendung wiederherstellen lässt.

```
1  PersistenceStore ps = new InMemoryPersistenceStore();
2  SiddhiManager sm = new SiddhiManager();
3  sm.setPersistenceStore(ps);
4
5  String siddhiApp = "..."; // beliebige Ereignisregel und Definition des TempEventStreams
6
7  StreamCallback qc = new StreamCallback() {
8      @Override
9      public void receive(Event[] events) {...} // beliebiger Methodenaufruf oder Ausgabe
10 };
11 SiddhiAppRuntime sar = sm.createSiddhiAppRuntime(siddhiApp);
12 sar.addCallback("query", qc);
13
14 InputHandler ih = sar.getInputHandler("TempEventStream");
15 sar.start();
16
17 ih.send(new Object[]{"sensor1", 1, 19.4});
18 ih.send(new Object[]{"sensor2", 3, 24.7});
19
20 sar.persist();
21
22 sar.shutdown();
23 sar = sm.createSiddhiAppRuntime(siddhiApp);
24 sar.addCallback("query", qc);
25 ih = sar.getInputHandler("TempEventStream");
26 sar.start();
27
28 sar.restoreLastRevision();
29
30 ih.send(new Object[]{"sensor2", 3, 24.4});
31 ih.send(new Object[]{"sensor3", 5, 18.1});
32
33 sar.shutdown();
```

Auflistung 5.27: Wiederherstellung des letzten Verarbeitungszustands einer Siddhi-Anwendung

Zuerst wird ein `PersistenceStore`-Objekt (Zeile 1) erzeugt und beim `SiddhiManager` registriert (Zeile 3), um die später generierten Snapshots zu speichern (In-Memory). Nach

Start der `SiddhiAppRuntime` werden zwei `TempEvents` über den `InputHandler` geschickt und anschließend ein Snapshot des Zustands mit der von `SiddhiAppRuntime` verfügbaren `persist`-Methode erstellt (Zeile 20). Die `SiddhiAppRuntime` wird beendet und kurz darauf neu erstellt. Die neu erstellte `SiddhiAppRuntime` kann gestartet werden und den letzten Zustand aus dem Speicher holen, indem die `restoreLastRevision`-Methode aufgerufen wird (Zeile 28). Wurde beispielsweise die Ereignisanfrage mit einem Window realisiert, würde Siddhi das Window mit den letzten beiden `TempEvents` wiederherstellen und anschließend weitere eintreffende `TempEvents` verarbeiten. An dieser Stelle sei erwähnt, dass trotz dieses Wiederherstellungsmechanismus Ereignisse verloren gehen können, falls Ereignisse kurz nach Ausfall eintreffen.

Eine weitere Möglichkeit zur Verbesserung der Fehlertoleranz, wäre die besprochene Skalierung in Abschnitt 5.2.1 von mehreren EPA-Instanzen. So können beispielsweise passive EPA-Instanzen auf einem Rechnerknoten bereitgestellt werden, welche ebenfalls Ereignisse einspeisen aber keine Ausgangsereignisse erzeugen sondern passiv den Verarbeitungszustand beibehalten. So kann bei Ausfall einer aktiven EPA-Instanz, eine passive Instanz einspringen. Jedoch ist für diese Realisierung einiges an zusätzlichem Aufwand gefordert, da untereinander auch Synchronisationszustände (bspw. durch Snapshots) ausgetauscht werden müssen, damit aktive und passive Instanzen den gleichen Verarbeitungsstand besitzen. [WSO18b]

Esper

Anders als Siddhi bietet die frei verfügbare Esper-Version leider keine Wiederherstellungsmechanismen oder Zustandspersistenz an. Solche Zustände lassen sich nur mit der Esper Enterpise Edition[4] im Zusammenhang mit Esper High Availability wiederherstellen. Da die Esper Enterpise Edition nicht Teil der Arbeit ist, wird an dieser Stelle nicht weiter darauf eingegangen. [Esp18e]

5.2.4. Installation & Integration

Anhand des Kriteriums *Installation & Integration* sollen die Installations- und Integrationsmöglichkeiten von Siddhi und Esper in eine Entwicklungsumgebung verglichen werden.

Siddhi

Für Siddhi gibt es drei Installations- und Integrationsmöglichkeiten in eine Entwicklungsumgebung, die im Folgenden vorgestellt werden.

[4] Weitere Informationen zur Esper Enterpise Edition unter http://www.espertech.com/data-sheets/

Bei der Umsetzung der Fallstudie wurde Siddhi problemlos in ein Java-Programm eingebettet. Um Siddhi zu benutzen, wurden die entsprechenden Softwareabhängigkeiten in der pom.xml eines Maven-Projekts eingebunden. Dennoch gibt es zwei weitere Alternativen, um die CEP Engine Siddhi zu benutzen.

Die erste Alternative wäre über ein IntelliJ-Plugin namens *Siddhi-Plugin-IDEA*, welches von WSO2 bereitgestellt wird. Mit dem Siddhi-Plugin-IDEA[5] wird eine Siddhi-Anwendung nur mittels Siddhi Streaming SQL implementiert und benötigt daher kein weiteren Java-Code. Das Plugin kann entweder direkt in IntelliJ-IDEA vom Jetbrains-Plugin-Repository oder mit der heruntergeladenen Plugin-Zip-Datei installiert werden. Zusätzlich muss ein *Siddhi Software Development Kit* (SDK) heruntergeladen werden, welches anschließend bei Erstellung eines Siddhi-Projektes hinzugefügt wird. [WSO18i]

Eine weitere Option, um Siddhi zu verwenden, bietet der bereits in Kapitel 3 kurz erwähnte WSO2 CEP. Da Siddhi als Kernengine im WSO2 CEP eingebettet ist, muss nach Installation des WSO2 CEP keine weitere Einbindung erfolgen. Der WSO2 CEP bietet das WSO2 Complex Event Processor Studio an, welches als Entwicklungsumgebung zur Erstellung von Siddhi-Anwendungen dient. Dadurch werden Siddhi-Anwendungen über einen Browser-Editor umgesetzt und ebenfalls nur mit der Siddhi Streaming SQL implementiert. Zusätzlich enthält der WSO2 CEP eine Menge an integrierten Werkzeugen und Features, die in Abschnitt 5.3.4 weiter diskutiert werden. [WSO18m]

Esper

Esper lässt sich als Bibliothek in ein Java-Projekt integrieren. Bei der Umsetzung der Fallstudie wurde Esper auch mit den entsprechenden Softwareabhängigkeiten in der pom.xml eines Maven-Projektes eingebunden. Eine weitere frei verfügbare Alternative ist NEsper[6], wodurch sich Esper mit dem .NET Framework von der Microsoft Software-Plattform .NET verwenden lässt.

5.3. Vergleich weiterer Kriterien

5.3.1. Benutzbarkeit & Wartbarkeit

Die beiden Kriterien *Benutzbarkeit & Wartbarkeit* sollen zeigen, ob hohe Aufwandskosten in die Einarbeitung der Ereignisabfragesprache entstehen und sich Änderungen am Code schnell und einfach umsetzen lassen.

[5] Weitere Informationen zum Siddhi-Plugin-IDEA unter https://wso2.github.io/siddhi-plugin-idea/

[6] Weitere Informationen zu NEsper unter http://www.espertech.com/esper/nesper-net/

Siddhi

Siddhis Ereignisanfragesprache hat den Vorteil, dass sie auf der deklarativen Datenbankanfragesprache SQL basiert und somit Konzepte realisiert, welche Benutzer bereits kennen. Aus diesem Grund haben Benutzer kaum Einstiegsschwierigkeiten bei der Umsetzung mit der Siddhi Streaming SQL. Noch dazu ist der Aufbau jeder Siddhi-Anwendung nach [(<Ereignisstromdefinition>|<Tabellendefinition>|<Windowdefinition>)+ → <Ereignisregel>+] besonders schematisch, sodass Änderungen unmittelbar an der richtigen Stelle vorgenommen werden können. Da Siddhi einige hilfreiche Aggregationsoperatoren anbietet, bleibt sogar die Erstellung von komplexen Formeln für spezifische Berechnungen wie Durchschnitt oder Standardabweichungen (siehe Abschnitt 5.1.1) erspart und benötigen keinen zusätzlichen Aufwand.

Ein vorteilhafter Wartungsaspekt ergibt sich durch die Siddhi-Architektur, die es ermöglicht, eine direkte Bearbeitung von Ereignisregeln während der Ausführung der Siddhi Engine vorzunehmen, um so Ereignisregeln hinzuzufügen, zu entfernen oder anzupassen. Jedoch ist die Wartbarkeit davon abhängig, in welcher Umgebung Siddhi eingebettet ist. In Kapitel 4 wurde Siddhi in ein Java-Programm eingebunden, wodurch die Definitionen der Ereignisströme und Ereignisregeln als String umgesetzt werden. Diese Umsetzung kann sich für größere Siddhi-Anwendungen mit sehr komplexen Ereignisregeln als wartungstechnischer Albtraum entwickeln, da sich die Übersicht im Gegensatz zu der Verwendung von Siddhi mit dem WSO2 CEP oder dem Siddhi-Plugin-IDEA (siehe Abschnitt 5.2.4) deutlich verschlechtern kann.

Esper

In Esper lassen sich die Ereignisregeln ebenso wie in Siddhi mühelos und verständlich definieren. Im Gegensatz zu Siddhi ist in einer Esper-Anwendung einiges mehr an Java-Implementierung gefragt, z. B. die Definition der Ereignistypen durch POJO-Klassen oder entsprechende Konfigurationen über die API-Klassen bei Umstellung des Zeitkonzepts (siehe Abschnitt 5.1.3) oder die Erstellung der Datenbankverbindung (siehe Abschnitt 5.1.5).

5.3.2. Dokumentation

Das Kriterium *Dokumentation* soll die beiden Dokumentationen der beiden Engines gegenüberstellen und auf Übersichtlichkeit, Strukturiertheit und enthaltene Code-Beispiele untersuchen.

Siddhi

Die Siddhi Dokumentation der Siddhi Streaming SQL ist äußerst übersichtlich und gut durchstrukturiert. Für alle Aggregationsfunktionen, Windows, Source und Sink Mapper, Tabellen, Siddhi-Erweiterungen und weitere Funktionen gibt es einige hilfreiche Implementierungsbeispiele. Das Gleiche gilt für komplexere Ereignisanfragen, die eine oder mehrere beispielhafte Ereignisregeln für jede verfügbare Funktion umsetzen. Dennoch fehlen wichtige Informationen für Multithreading von Ereignisströmen, welche an dieser Stelle besonders wünschenswert wären. Ebenso fehlt es in der API Dokumentation an Beschreibung der Klassen und deren vorhandenen Methoden. In diesem Fall muss für die beiden wichtigen Klassen `SiddhiManager` und `SiddhiAppRuntime` auf Siddhis Quellcode zurückgegriffen werden, was sehr mühsam und aufwändig ist. Trotz allem existiert eine detaillierte Architekturbeschreibung von Siddhi, die Entwicklern eine verständnisvollere Sicht der Abläufe innerhalb einer Siddhi-Anwendung übermittelt.

Esper

Esper bietet ebenso eine gut durchstrukturierte Dokumentation an, ist jedoch im Vergleich zu Siddhi um einiges umfangreicher. Alle Sprachelemente, Funktionen und APIs werden in Esper ausführlich erklärt. Die Dokumentation enthält komplexe Beispielregeln, welche einige Ereignismuster für verschiedene aufschlussreiche Situationen formulieren. Espers Dokumentation fehlt es durchweg an nichts und umfasst sogar kurze Beschreibungen der in Esper umgesetzten Fallstudien, die im Esper Softwarepaket enthalten sind und verschiedene Szenarien behandeln.

5.3.3. Weiterentwicklung der Engine

Das Kriterium *Weiterentwicklung der Engine* soll untersuchen, ob Siddhi und Esper noch in Zukunft weiter unterstützt werden.

Siddhi

Siddhi veröffentlichte während der Arbeit die Version 4.3.0, die sich noch weiter in aktiver Entwicklung befindet. Da hinter Siddhi der Open-Source-Technologieanbieter WSO2 steckt und dieser Siddhi für einige seiner Softwareprodukte nutzt, kann davon ausgegangen werden, dass Siddhi auch in Zukunft weiter entwickelt und unterstützt wird.

Esper

Während der Arbeit veröffentlichte auch EsperTech am 23. Oktober 2018 eine neue Esper-Version 8.0.0. [Esp18d] Der wesentliche Unterschied zwischen der alten und neuen Esper-Version steckt in der Architektur. Im Gegensatz zur Version 7.1.0 oder älteren Versionen, die eine monolithische Interpreterarchitektur besitzen, enthält Version 8 eine neue Compiler- und Laufzeitarchitektur. In diesem Zusammenhang bedeutet dies, dass Esper seit Version 8 nicht nur eine eigene Sprache besitzt, sondern auch einen eigenen Sprachcompiler und eine eigene Laufzeitumgebung. [Esp18b] Zusätzliche Änderungen an der EPL wurden in der neuen Version jedoch nicht vorgenommen. Bei weiterem Interesse an den Änderungen zwischen Esper-Version 7.1.0 und 8.0.0 können diese unter der Quelle [Esp18c] nachgelesen werden. Esper wird daher ohne Bedenken auch in Zukunft weiter unterstützt und lässt noch einiges an Änderungen mit sich bringen.

5.3.4. Weitere Features & Werkzeuge

Das Kriterium *Weitere Features & Werkzeuge* soll zeigen, welche weiteren Features und Werkzeuge die beiden Engines bieten, z. B. Dashboards, Debugger-Tools etc.

Siddhi

Siddhi stellt zurzeit 55 Erweiterungen bereit. Jede Erweiterung enthält für sich spezifische Funktionen, die in Siddhis Ereignisregeln benutzt werden können. So verfügt beispielsweise die *siddhi-execution-math*-Erweiterung einige mathematische Funktionen, um die Ereignisregeln flexibler zu gestalten. Außerdem bietet die Erweiterung *siddhi-script-js* die Einbettung von JavaScript-Funktionen innerhalb der Siddhi Streaming SQL. [WSO18f] Siddhi bietet dem Benutzer als weiteres Feature das Formulieren von eigenen Funktionen an. Mithilfe der *FunctionExecuter*-Klasse können die benutzerdefinierten Funktionen implementiert und in Siddhi Ereignisregeln verwendet werden. Außerdem können mit der `@app:statistics`-Annotation Statistiken der Siddhi-Anwendung ausgewertet werden. Diese lassen sich in festgelegten Zeitintervallen auf der Konsole ausgeben und führen verschiedene Messwerte der Anwendungskomponenten (z. B. Stream, Query, Window, Mapper, ...) auf. [WSO18l]

In Abschnitt 5.2.4 wurde bereits erwähnt, dass sich Siddhi mit dem Siddhi-Plugin-IDEA oder mit dem WSO2 CEP verwenden lässt. Wird Siddhi mit dem Siddhi-Plugin-IDEA genutzt, enthält dieses Plugin einige nützliche Entwicklungswerkzeuge, die einem die Implementation und Ausführung einer Siddhi-Anwendung erleichtern. So können beispielsweise mit dem enthaltenen Debugger-Tool die Fehler einer Siddhi-Anwendung unkompliziert diagnostiziert und aufgefunden werden. Außerdem enthält das Plugin eine aktive Rechtschreibprüfung für alle Bezeichner der Siddhi-Anwendung und unterstützt das Hervorheben der Siddhi-Streaming-SQL-Syntax sowie die automatische Code-

Vervollständigung. Alle mit dem Plugin entwickelten Siddhi-Anwendungen werden als `.siddhi`-Datei gespeichert. Diese können in IntelliJ-IDEA ausgeführt werden, sodass per Konfiguration eingestellt werden kann, welche Datei mit enthaltenen Testdaten ausgelesen wird, z. B. eine Textdatei. Ebenso sind mit dem Plugin alle Erweiterungen von Siddhi einsetzbar. [WSO18i]

Der WSO2 CEP bietet ebenfalls eine ganze Reihe an nützlichen Werkzeugen und Features. Neben der Entwicklungsumgebung WSO2 Stream Processor Studio unterstützt der WSO2 CEP Monitoring- und Debugging-Tools. So lassen sich, z. B. Ereignisverarbeitungsabläufe mit zusätzlichen simulierenden Ereignisströmen visualisieren. Ebenso können aus Ereignisverarbeitungsabläufen Statistikdaten erstellt und durch eingebettete Dashboards in Echtzeit dargestellt werden. Außerdem kann mit dem Siddhi-Try-Out-Tool die Siddhi Streaming SQL vorweg getestet werden, um diese auf verschiedene Funktionen und Benutzbarkeit zu testen. Genauso können mit dem WSO2 CEP alle Erweiterungen verwendet werden, die bereits eingebettet sind. [WSO18m]

Esper

EsperTech bietet neben Esper auch die verteilbare Plattform Esper Enterpise Edition an. Die Plattform unterstützt ebenfalls einen EPL-Editor mit Debugger-Tool als Entwicklungsumgebung sowie das Führen von Statistikdaten über Zeitmessung der Ereignisverarbeitung und Speicherauslastung. Neben der linearen und horizontalen Skalierbarkeit und fehlertoleranten Ereignisverarbeitung (siehe Abschnitt 5.2.3) bietet die Plattform auch viele Erweiterungen bezüglich der Verwendung einiger Tools an, z. B. Dashboard- und Laufzeitverwaltungstools oder webbasierte Benutzeroberflächen zur Verwaltung von mehreren verteilten EPA-Instanzen. Esper Enterpise Edition ist jedoch ein kommerzielles Produkt für Geschäftskunden und daher nicht frei verfügbar. [Esp18e] Ähnlich zum Siddhi-Try-Out-Tool stellt auch Esper ein solches Tool über eine webbasierte Benutzeroberfläche[7] zur Verfügung, um Ereignisanfragen mit der Esper EPL zu testen.

5.4. Auswertung und Endergebnis

Siddhi und Esper wurden in den vorherigen Abschnitten nach den aufgestellten Evaluierungskriterien verglichen. Nun sollen die beiden CEP Engines anhand dieser Kriterien evaluiert werden. Zur Evaluierung wurde folgende Bewertungsskala genutzt:

[7] Esper EPL Online Tool unter http://esper-epl-tryout.appspot.com/epltryout/mainform.html

Bewertungsstufe	Beschreibung
+	Die CEP Engine erfüllt das Kriterium **und** zeigt qualitative Stärken.
○	Die CEP Engine erfüllt das Kriterium, hat jedoch geringe Schwächen.
−	Die CEP Engine erfüllt nicht das Kriterium **oder** weist erhebliche Schwächen auf.

Kriterium	Referenz	Bewertung Siddhi	Bewertung Esper
Sprachliche und konzeptionelle Kriterien			
Kernoperatoren & Aggregationsfunktionen	5.1.1	○	+
Windows	5.1.2	○	+
Zeitkonzept	5.1.3	○	+
Datenextraktion & Umwandlung verschiedener Datenformate	5.1.4	+	○
Ereignispersistenz	5.1.5	+	+
Behandlung von Ereignisduplikaten	5.1.6	+	+
Parallelisierung von Ereignisverarbeitung	5.1.7	○	+
Technische Kriterien			
Skalierbarkeit	5.2.1	○	○
Performanz	5.2.2	○	+
Fehlertoleranz	5.2.3	○	−
Installation & Integration	5.2.4	+	+
Weitere Kriterien			
Benutzbarkeit & Wartbarkeit	5.3.1	○	+
Dokumentation	5.3.2	○	+
Weiterentwicklung der Engine	5.3.3	+	+
Weitere Features & Werkzeuge	5.3.4	+	○

Tabelle 5.1.: Evaluierungstabelle

Die Tabelle 5.1 zeigt deutlich, dass Siddhi in einigen sprachlichen und konzeptionellen Kriterien nicht die gleiche Reife wie Esper erreicht. Siddhi besitzt zwar alle grundlegenden Operatoren, Aggregationsfunktionen und Windows, die sich in den Ereignisregeln einfach anwenden lassen, verfügt jedoch nicht über das breite Spektrum wie Esper. Daher wurde bei diesen beiden Kriterien nach der bereitgestellten Vielfalt an Funktionen, Operatoren und Windows bewertet.

Das Zeitkonzept in Siddhi lässt sich nur mithilfe der Zeitstempel von eintreffenden Ereignissen festlegen. In diesem Fall bietet Esper mehr Kontrolle, sodass sich die Zeit nicht nur über sendende Zeitereignisse steuern lässt, sondern auch anhand einer beliebigen Systemzeit bestimmt werden kann.

Die Datenextraktion und Umwandlung in verschiedene Datenformate ist in Siddhi durch Hinzufügen von Annotationen einfach gehalten und bietet umfangreiche Transportmöglichkeiten. Zwar kann Esper mehrere Datenformate der Ereignisse ohne vorheriger Umwandlung verarbeiten (z. B. XML), unterstützt jedoch deutlich weniger Transportadapter als Siddhi, sodass diese erst implementiert werden müssen.

Die Ereignispersistenz setzen beide CEP Engines durch Ereignistabellen um, die fast auf die gleiche Art und Weise angelegt werden. Ebenso lassen sich in beiden Fällen die Verbindung zu Datenbanken herstellen, sodass Ereignisse in Datenbank-Instanzen gespeichert werden können. Auch die Behandlung von Ereignisduplikaten führen beide Engines anhand von speziellen Unique-Windows durch. Aus diesem Grund erfüllen beide Engines die wünschenswerten Anforderungen der beiden Kriterien.

Bei der Parallelisierung der Ereignisse zeigt Esper deutlich mehr Möglichkeiten, die zwar in dieser Arbeit nicht alle erwähnt wurden, aber mehr Konfigurationen der Threads bietet. In Siddhi können nur drei Konfigurationsparameter der Annotation übergeben werden, aber keine weitere Ablaufkonfigurationen der Threads.

Bei den technischen Kriterien sind beide CEP Engine gleich gewichtet. Siddhi zeigte sich beim Performanztest in Abschnitt 5.2.2 überraschender Weise etwas schlechter. Dennoch hat der Performanztest in einigen Fällen ergeben, dass Siddhi frühzeitig mehr Ereignisse als Esper liefert. Allerdings sollte anstelle des einfachen Performanztests in dieser Arbeit, ein umfangreicherer Performanztest vollzogen werden, um eine aussagekräftige Entscheidung über die Leistung der beiden CEP Engines zu fällen.

Siddhi und Esper lassen sich beide horizontal skalieren. Dennoch ist die Skalierung der beiden Engines mit Aufwand verbunden, da der Benutzer den korrekten Kommunikationsablauf selbst implementieren muss, d. h. welcher EPA muss mit welchem Ziel-EPA interagieren.

Sollte es während der Ereignisverarbeitung zu einem Ausfall kommen, lässt sich der letzte Verarbeitungszustand in Siddhi durch periodisch erstellte Snapshots wiederherstellen. Im Allgemeinen gibt es solch ein Wiederherstellungsmechanismus in Esper nicht und wird daher den Anforderungen der Fehlertoleranz nicht gerecht.

Diverse Installations- und Integrationsmöglichkeiten in eine Entwicklungsumgebung bieten beide CEP Engines, daher erfüllt Siddhi als auch Esper dieses Kriterium.

Die Implementierung der Fallstudie mit Siddhi und Esper bewies sich in beiden Fällen als unkompliziert. Die Ereignisregeln sind mit beiden Ereignisanfragesprachen (EQL und Siddhi Streaming SQL), durch die SQL-angelehnte Syntax, schematisch aufgebaut und deshalb einfach zu definieren. Die Siddhi Streaming SQL als String eingebettet in Java, führt jedoch schnell zur Unübersichtlichkeit. Sind beispielsweise Änderungen des Ereignistyps einer Ausgangsstromdefinition nötig, kann dies zu weiteren Änderungen an den Eingangsstromdefinitionen in anderen EPA-Instanzen mit sich tragen. Dies ist in Esper durch beispielsweise Java-Objekte besser gelöst.

Die Siddhi Dokumentation ist gut strukturiert, enthält aber zu einigen wichtigen Aspekten, die in Abschnitt 5.3.2 aufgeführt wurden, keine ausgiebigen Informationen. Esper hat hingegen eine sehr ausführliche und ausgedehnte Dokumentation.

Während der Arbeit wurden neue Siddhi- und Esper-Versionen veröffentlicht, die sich beide momentan in aktiver Entwicklung befinden. Daher ist auch in Zukunft die weitere Unterstützung der beiden Engines zu erwarten.

Im Gegensatz zu Esper, welches nur die kommerzielle Esper Enterprise Edition anbietet, kann Siddhi mit dem frei verfügbaren WSO2 CEP verwendet werden. Wie bereits in Abschnitt 5.3.4 besprochen, enthält der WSO2 CEP einige hilfreiche Tools und durch die bereitgestellte Entwicklungsumgebung lassen sich die Siddhi-Anwendungen nur mit der Siddhi Streaming SQL implementieren, welches an zusätzlichen Java-Code erspart.

6. Fazit

In dieser Arbeit wurde die CEP Engine Siddhi vorgestellt. Anhand von ausgewählten Evaluerierungskriterien wurde Siddhi mit der etablierten CEP Engine Esper auf sprachlicher, konzeptioneller und technischer Ebene verglichen. Ebenso wurde die praktische Implementierung einer Fallstudie jeweils mit Siddhi und Esper umgesetzt. Beide CEP Engine verfolgen das gleiche Konzept und verwenden als Ereignisanfragesprache die Datenstrom-Anfragesprache CQL. Die Evaluierung der beiden Engines hat ergeben, dass Siddhi alle Kriterien mit den grundlegenden Anforderungen erfüllt und zeigt nur in einigen Aspekten leichte schwächen zu Esper, die in Zukunft auf zuversichtliche Verbesserungen warten lassen. Grundsätzlich zeigt Siddhi einiges an Potenzial, vor allem durch die einfache Benutzung der Siddhi Streaming SQL und den nützlichen Siddhi-Erweiterungen. Anhand des übersichtlichen Umfangs dient Siddhi als gute CEP Einstiegs-Engine für Benutzer, die sich erst neu mit dem Thema Complex Event Processing auseinander gesetzt haben. Außerdem bietet Siddhi in Kombination mit dem frei verfügbaren WSO2 CEP besonders interessante Möglichkeiten in einigen Bereichen, z. B. wird der WSO2 CEP erfolgreich auf einer *Clinical Intelligence Platform* im amerikanischen Krankenhaus *Cleveland Clinic* eingesetzt, um Gesundheitsdaten in Echtzeit zu analysieren und diese auch zu visualisieren und zu überwachen. [WSO18m]

Am Ende der Arbeit lässt sich schwer eine Entscheidung treffen, welche CEP Engine als empfehlenswerter erscheint. Die Fallstudie hat ergeben wie ähnlich sich die Ereignisregeldefinitionen der beiden Engines umsetzen lassen, sodass zumindest an diesem Punkt keine eindeutige Wahl getroffen werden kann. Nach den Evaluierungskriterien hat sich jedoch Esper als die bessere CEP Engine hervorgehoben. Durch die zwölf produktiven Jahre der Engine Esper ist dementsprechend ein umfangreiches Spektrum herangewachsen mit etlichen ausgereiften Funktionen für den erfolgreichen Einsatz in CEP-Projekten. Dennoch ist Siddhi aufgrund der Bewertung nicht ganz auszuschließen und kann durchaus in vielen Anwendungen als konstruktive CEP Engine in Betracht gezogen werden.

Literaturverzeichnis

[Bae18] Baeldung: *Concurrency with LMAX Disruptor – An Introduction*. Website, 2018. – https://www.baeldung.com/lmax-disruptor-concurrency; abgerufen am 07. Dezember 2018.

[BD10] Bruns, R. ; Dunkel, J.: *Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Berlin, Heidelberg : Springer Verlag, 2010

[BD15] Bruns, R. ; Dunkel, J.: *Complex Event Processing: Complexe Analyse von massiven Datenströmen mit CEP*. Wiesbaden : Springer Vieweg, 2015

[Dah18] Dahanayakage, T.: *Incorporating Siddhi Query Runtime in a Multithreading Environment*. Website, 2018. – https://stackoverflow.com/questions/50943761/incorporating-siddhi-query-runtime-in-a-multithreading-environment; abgerufen am 07. Dezember 2018.

[EBle] Eckert, M. ; Bry, F.: Complex Event Processing (CEP). In: *Informatik Spektrum* 32 (2009, Seattle), S. 163–167

[Esp18a] EsperTech Inc.: *Costumers*. Website, 2018. – http://www.espertech.com/customers/; abgerufen am 01. November 2018.

[Esp18b] EsperTech Inc.: *Esper 8 Compiler und Runtime*. Website, 2018. – http://www.espertech.com/2018/10/23/esper-8-compiler-and-runtime/; abgerufen am 01. November 2018.

[Esp18c] EsperTech Inc.: *Esper-8 Conceptual Differences to Older Versions*. Website, 2018. – http://www.espertech.com/2018/10/23/esper-8-conceptual-differences-to-older-versions/; abgerufen am 01. November 2018.

[Esp18d] EsperTech Inc.: *Esper Change History*. Website, 2018. – http://www.espertech.com/esper/esper-changehistory/; abgerufen am 01. November 2018.

[Esp18e] EsperTech Inc.: *Esper Enterprise Edition*. Website, 2018. – http://www.espertech.com/esper-enterprise-edition/; abgerufen am 01. November 2018.

[Esp18f] EsperTech Inc.: *Esper Reference Version 7.1.0*. Website, 2018. – http://esper.espertech.com/release-7.1.0/esper-reference/html/index.html; abgerufen am 10. Oktober 2018.

[Esp18g] EsperTech Inc.: *EsperIO Reference Version 7.1.0*. Website, 2018. – http://esper.espertech.com/release-7.1.0/esperio-reference/html_single/index.html#adapter_overview; abgerufen am 07. Dezember 2018.

[Esp18h] EsperTech Inc.: *License and Trademark Information*. Website, 2018. – http://www.espertech.com/esper/esper-license-and-trademark/; abgerufen am 01. November 2018.

[Hed17] Hedtstück, U.: *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. Berlin : Springer Vieweg, 2017

[Int11] International Organization for Standardization: *ISO-Store ISO/IEC 25010:2011*. Website, 2011. – https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en; abgerufen am 23. September 2018.

[Luc02] Luckham, David C.: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002

[Robnd] Robins, D. B.: Complex Event Processing. In: *CSEP 504* (2010, Redmond)

[SGN+le] Suhothayan, S. ; Gajasinghe, K. ; Narangoda, I. L. ; Chaturanga, S. ; Perera, S. ; Nanayakkara, V.: Siddhi: A second look at complex event processing architectures. In: *Proceedings of the ACM workshop on Gateway computing environments* (2011, Seattle), S. 43–50

[Suh18] Suhothayan, S.: *TimestampGeneratorImpl-Klasse zur Erzeugung von Zeitstempel*. Website, 2018. – https://github.com/wso2/siddhi/blob/master/modules/siddhi-core/src/main/java/org/wso2/siddhi/core/util/timestamp/TimestampGeneratorImpl.java; abgerufen am 07. Dezember 2018.

[Vin10] Vincent, Paul: *TIBCO blog - Complex Event Processing: a technology evaluation check-list*. Website, 2010. – https://www.tibco.com/blog/2010/03/04/complex-event-processing-a-technology-evaluation-check-list/; abgerufen am 21. September 2018.

[WSO18a] WSO2: *API Docs - v4.2.31*. Website, 2018. – https://wso2.github.io/siddhi/api/4.2.31/; abgerufen am 03. November 2018.

[WSO18b] WSO2: *Clustering CEP 4.1.0 and 4.2.0*. Website, 2018. – https://docs.wso2.com/display/CLUSTER44x/Clustering+CEP+4.1.0+and+4.2.0; abgerufen am 10. Dezember 2018.

[WSO18c] WSO2: *Siddhi*. Website, 2018. – https://wso2.github.io/siddhi/; abgerufen am 21. September 2018.

[WSO18d] WSO2: *Siddhi Architecture*. Website, 2018. – https://wso2.github.io/siddhi/documentation/siddhi-architecture/; abgerufen am 21. September 2018.

[WSO18e] WSO2: *Siddhi Execution Unique*. Website, 2018. – https://wso2-extensions.github.io/siddhi-execution-unique/; abgerufen am 07. Dezember 2018.

[WSO18f] WSO2: *Siddhi Extensions*. Website, 2018. – https://wso2.github.io/siddhi/extensions/; abgerufen am 21. September 2018.

[WSO18g] WSO2: *Siddhi IO RabbitMQ Erweiterung*. Website, 2018. – https://wso2-extensions.github.io/siddhi-io-rabbitmq/; abgerufen am 07. Dezember 2018.

[WSO18h] WSO2: *Siddhi IO Websocket Erweiterung*. Website, 2018. – https://wso2-extensions.github.io/siddhi-io-websocket/api/1.0.11/; abgerufen am 07. Dezember 2018.

[WSO18i] WSO2: *Siddhi-Plugin-IDEA*. Website, 2018. – https://wso2.github.io/siddhi-plugin-idea/; abgerufen am 01. November 2018.

[WSO18j] WSO2: *Siddhi Source Code - Interface-Klasse PersistenceStore*. Website, 2018. – https://github.com/wso2/siddhi/blob/master/modules/siddhi-core/src/main/java/org/wso2/siddhi/core/util/persistence/PersistenceStore.java; abgerufen am 07. Dezember 2018.

[WSO18k] WSO2: *Siddhi Store Cassandra Erweiterung*. Website, 2018. – https://wso2-extensions.github.io/siddhi-store-cassandra/; abgerufen am 07. Dezember 2018.

[WSO18l] WSO2: *Siddhi Streaming SQL Guide 4.0*. Website, 2018. – https://wso2.github.io/siddhi/documentation/siddhi-4.0/; abgerufen am 21. September 2018.

[WSO18m] WSO2: *WSO2 Complex Event Processor*. Website, 2018. – https://wso2.com/products/complex-event-processor/; abgerufen am 21. September 2018.

[YCL11] Yao, W. ; Chu, C. ; Li, Z.: Leveraging complex event processing for smart hospitals using RFID. In: *Journal of Network and Computer Applications* 34 (3) (2011), S. S. 799–810

A. Inhalt des Datenträgers

Die beigefügte CD enthält den nachfolgenden Inhalt:

- Digitale Version der Bachelorarbeit
- Quellcode-Dateien
  - Implementierung der Fallstudie mit Siddhi
  - Implementierung der Fallstudie mit Esper